Universiteit Gent
Faculteit Toegepaste Wetenschappen
Faculteit Wetenschappen
Vakgroep
Zuivere Wiskunde en Computeralgebra
Voorzitter: Prof. A. Hoogewijs

Uitbreiding van de Model Specification Language (MSL) compiler

door
Thierry CORNELIS

Promotor: Prof. Albert Hoogewijs
Co-promotoren:
Prof. Peter Vanrolleghem
Prof. Hans Vangheluwe

Scriptie neergelegd tot het behalen van de academische graad van Licentiaat in de Informatica

Academiejaar 2002-2003

Woord vooraf

Deze scriptie werd neergelegd tot het behalen van de graad van Licentiaat in de Informatica.

Na een diploma Kandidaat Bio-Ingenieur behaald te hebben zocht ik andere oorden op en begon aan de studies informatica. Toch was ik de smaak van de biologische wetenschappen nog niet helemaal kwijt en informeerde ik bij Ingmar Nopens, doctoraalstudent aan de vakgroep "Toegepaste wiskunde, biometrie en procesregeling" (Biomath) van de Universiteit Gent, of er een mogelijkheid bestond een thesis te maken in samenwerking met deze vakgroep. Prof. Vanrolleghem was hierover zeer positief en gaf de goedkeuring om te werken aan een nieuwe MSL compiler. Bij deze wil ik dan ook Prof. Vanrolleghem bedanken voor de mogelijkheid een thesis te maken in samenwerking met Biomath.

Om de gebruikte software te leren kennen werd een verblijf van zes weken gepland in Montreal, Canada, aan de McGill University. Daar werd ik begeleid door Prof. Vangheluwe, hoofd van het MSDL (Modelling, Simulation and Design). Een hartelijke dank gaat, naast Prof. Vangheluwe, ook uit naar de andere MSDL-medewerkers: Indrani A.V., Ernesto Posse, Jean-Sébastien Bolduc, Spencer Borland, Marc Provost, Simon Lacoste-Julien en Wei Bin Liang.

Filip Claeys, Software engineer, modeling & simulation bij HEMMIS te Kortrijk en Dirk De Pauw, doctoraalstudent aan de vakgroep Biomath, stonden in voor de begeleiding van mijn thesis. Hierbij wil ik hen dan ook van harte bedanken voor de veelvuldige en duidelijke uitleg.

Ook een hartelijke dank aan Prof. Hoogewijs die promotor wou zijn van deze thesis.

Daarnaast wil ik mijn vriendin bedanken voor het talrijke nalezen van mijn thesis.

Ten slotte wil ik mijn ouders en vrienden bedanken die steun hebben geboden bij het tot stand komen van deze thesis.

29 mei 2003,

Uitbreiding van de Model Specification Language (MSL) compiler
door
Thierry CORNELIS

Scriptie neergelegd tot het behalen van de academische graad van Licentiaat in de Informatica

Academiejaar 2002-2003

Promotor: Prof. Albert Hoogewijs
Co-promotoren:
Prof. Peter Vanrolleghem
Prof. Hans Vangheluwe

Faculteit Toegepaste Wetenschappen
Faculteit Wetenschappen
Universiteit Gent

Vakgroep Zuivere Wiskunde en Computeralgebra
Voorzitter: Prof. A. Hoogewijs

Samenvatting

De MSL compiler wordt door de modelleer- en experimenteer-omgeving WEST++ gebruikt om modellen in MSL-USER om te zetten naar MSL-EXEC (C++ code). De bestaande MSL compiler heeft bepaalde beperkingen en is bijgevolg aan vervanging toe.

In hoofdstuk één wordt de structuur van een compiler geschetst en worden de verschillende stappen van het compilatieproces besproken.

Hoofdstuk twee handelt over AToM^3, een meta-modelleer-omgeving geschreven in Python, waarin de MSL compiler zal geïntegreerd worden. Voor de integratie van de MSL compiler wordt deze als Python-extensie met een callback mechanisme gebruikt. Deze twee technieken worden in dit hoofdstuk behandeld.

In het derde hoofdstuk wordt de modelleer- en experimenteer-omgeving WEST++ geïntroduceerd. Ook de MSL-USER taal wordt in dit hoofdstuk besproken.

De bespreking van de MSL compiler is te vinden in hoofdstuk vier. De compiler wordt opgebouwd met behulp van een aantal prototypes. Een eerste prototype introduceert de

tools flex en bison. Het tweede protototype maakt gebruik van een Python-extensie. Deze twee worden gecombineerd om de MSL compiler te bouwen. Er wordt dan een MSL USER voorbeeld besproken dat wordt gecompileerd naar MSL-EXEC. De MSL-EXEC code leidt tot een simulatie in WEST++.

Trefwoorden : compiler, modelleren, meta-modelleren, WEST++, $AToM^3$

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Inleiding

Mijn doelstelling bestond uit het ontwikkelen van een nieuwe MSL compiler voor de MSL-USER 4.0 taal. De compiler dient integreerbaar te zijn met het programma AToM^3. De compiler dient eveneens een uitgebreide functionaliteit te hebben. Zo dient naast het kunnen herkennen van correcte MSL-USER 4.0 code een stelsel van vergelijkingen vertaald te kunnen worden naar MSL-EXEC en dient lexical scoping aanwezig te zijn.

Voor de theorie aangaande compilers heb ik mij vooral gebaseerd op het boek *Compilers - Principles, Techniques and Tool* van *AlfredV. Aho*, *Ravi Sethi* en *Jeffrey Ullman*.
Om meer inzicht te krijgen in AToM^3 heb ik verschillende artikels van *Prof. Vangheluwe* geraadpleegd.
Het implementatiegedeelte van deze thesis vond plaats in Python, C en C++. De boeken *Python 2.1 Bible* van *Dave Brueck en Tanner Stephen*, *A Book on C* van *Al Kelley en Pohl Ira* en *The C++ Programming Language* van *Bjarne Stroustrup* werden dan ook vaak ter hand genomen.
Internet, en meer specifiek de Python-[1] en AToM^3-website [2], is vanzelfsprekend niet weg te denken als bron van informatie.

Om tot de bouw van een compiler te komen dienen eerst de principes duidelijk te worden weergegeven. Zo worden in het eerste hoofdstuk de fasen van het compilatieproces besproken: de lexicale analyse, de grammaticale analyse, de semantische analyse, de intermediaire code generatie, de code optimalisatie en code generatie. Het compilatieproces wordt daarvoor opgedeeld in twee delen: de *front-end* en de *back-end*. Naast de front-end en de back-end bouwt de compiler een *symbol table* en een *abstract syntax tree* op.

In hoofdstuk twee wordt de meta-modelleer-omgeving AToM^3 geïntroduceerd. Er wordt kort uitgelegd wat AToM^3 en een *meta-model* zijn. De taal waarin AToM^3 is

[1] http://www.python.org/
[2] http://atom3.cs.mcgill.ca/

geïmplementeerd is Python. Python heeft de mogelijkheid om een C/C++ functie op te roepen vanuit de Python-interpreter. Er wordt dan van een extensie gesproken. $AToM^3$ zal een dergelijke extensie samen met een callback mechanisme gebruiken om de MSL compiler op te roepen. Beide worden in hoofdstuk twee besproken.

In hoofdstuk drie wordt de simulatie- en experimenteer-omgeving WEST++ voorgesteld. In WEST++ worden grafische modellen vertaald naar MSL-USER. De WEST++ gebruikt de MSL compiler om MSL-USER om te zetten naar MSL-EXEC. WEST++ kan verschillende soorten virtuele experimenten uitvoeren uitgaande van MSL-EXEC code. De MSL-USER 4.0 syntax wordt in dit hoofdstuk beschreven.

Hoofdstuk vier bevat de beschrijving van de compilerimplementatie. Er wordt met prototypes gewerkt. Eerst wordt een kleine compiler gebouwd om de tools *flex* en *bison* onder de knie te krijgen. Daarna wordt dezelfde compiler als een Python-extensie geïmplementeerd voorzien van een callback mechanisme. Uiteindelijk worden de principes van de twee prototypes samengebracht in de MSL compiler.

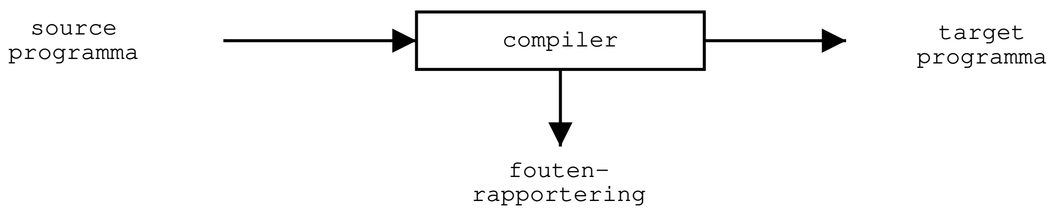
Hoofdstuk I

Compilers

1.1 Inleiding

Een compiler is voor de meeste computergebruikers totaal onbekend, wat begrijpelijk is aangezien ze er nooit gebruik van dienen te maken en er zeker zelf geen ontwikkelen. Toch zouden de toepasssingen die ze aanwenden niet bestaan indien er geen compilers zouden zijn.
De meeste programmeertalen (b.v. C en C++) zijn talen die gecompileerd worden tot een uitvoerbaar programma. Het zou niet mogelijk zijn de computertoepassingen die vandaag de dag ontwikkeld worden zonder compiler te vertalen naar uitvoerbare code. Daarvoor zijn ze namelijk te complex.
Het voorhanden zijn van een compiler voor een taal is bijgevolg duidelijk van groot belang.

In het eerste hoofdstuk wordt een overzicht gegeven van de diverse onderdelen van een compiler. Er wordt een onderscheid gemaakt tussen de verschillende fasen waaruit een compilatieproces bestaat.
De fasen worden vervolgens gegroepeerd in een *front-end* en een *back-end*. Verder in dit hoofdstuk worden de lexicale, grammaticale en semantische analyse en intermediaire code generatie, die samen de front-end vormen, behandeld. Tevens worden de code optimalisatie en code generatie, die samen de back-end van de parser vormen, ook in dit hoofdstuk besproken. De symbol table wordt tijdens alle fasen van het compilatieproces bewerkt of bevraagd maar wordt in de context van de front-end toegelicht omdat de opbouw daar een aanvang neemt.



Figuur 1.1: Compiler

1.2 Definitie

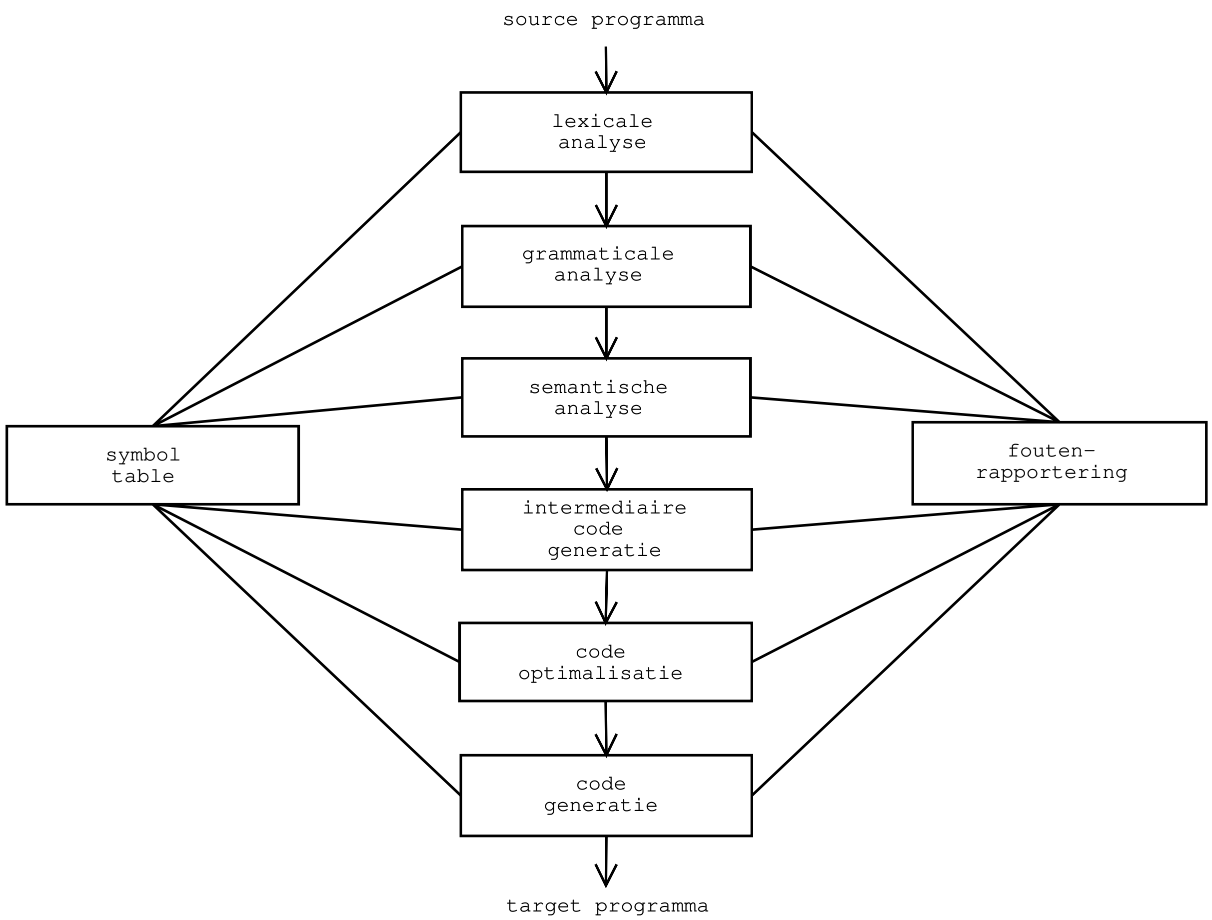
Compiler : Een *compiler* is een programma dat een programma geschreven in een bepaalde taal, de *source* taal, leest en vertaalt naar een equivalent programma in een andere taal, de *target* taal (figuur 1.1) [2].

In figuur 1.1 is tevens een onderdeel foutenrapportering opgenomen. Het spreekt voor zich dat dit een belangrijk onderdeel van de compiler vormt.
Fouten in het source programma dienen aan de gebruiker gemeld te worden zodat deze de gepaste actie kan ondernemen. Foutenrapportering dient ruimer gezien te worden dan enkel het naar het scherm schrijven van het lijn- en kolomnummer. De gebruiker is namelijk niet beperkt tot een fysiek persoon maar kan een ander programma zijn dat de compiler oproept. Een programma verwacht een return waarde en geen uitvoer naar het scherm.

1.3 Fasen van het compilatieproces

Een compilatieproces kan opgedeeld worden in verschillende *fasen* die elkaar logisch opvolgen.
Een fase heeft de eigenschap dat ze het source programma van de ene naar de andere voorstelling transformeert. De verschillende fasen zijn opgenomen in figuur 1.2. Het is niet altijd zo dat alle fasen op zichzelf bestaan. Sommige fasen kunnen gegroepeerd worden en de verschillende tussentijdse voorstellingen van het source programma worden niet altijd expliciet geconstrueerd.
Er bestaan ook andere opdelingen in fasen van het compilatieproces [25] dan diegene die in figuur 1.2 aan bod komen.

De opdeling van het compilatieproces in fasen is een opsplitsing in logisch verschillende onderdelen. Bij de implementatie van een compiler worden logisch verschillende fasen vaak



Figuur 1.2: Fasen van het compilatieproces [2]

samen genomen tot één geheel. Er wordt dan een opdeling gemaakt in een *back-end* en een *front-end*.

De front-end van een compiler bestaat uit de fasen die hoofdzakelijk afhankelijk zijn van de source taal en onafhankelijk van de target machine [2]. Tot de front-end behoren bijgevolg de lexicale analyse, de grammaticale (of syntax) analyse, de semantische analyse, het opbouwen van de symbol table en de generatie van intermediaire code. Bij elke fase of gecombineerde fasen van de front-end hoort een foutenrapportering die bijgevolg deel uitmaakt van de front-end.

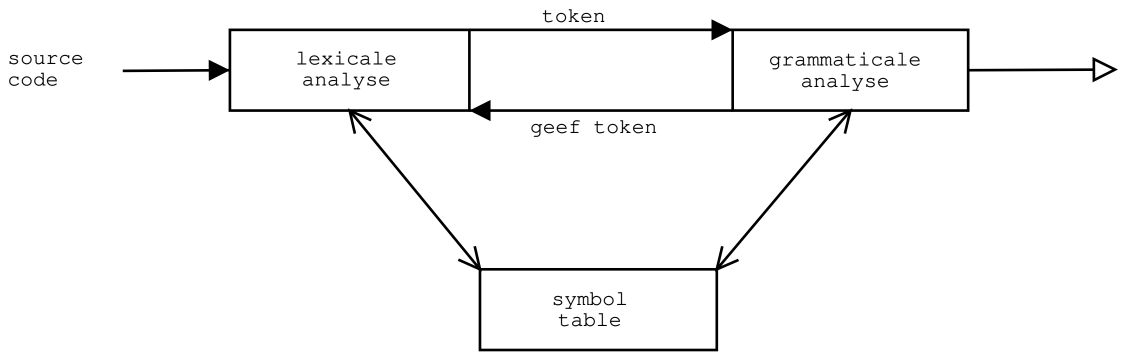
De back-end van een compiler omvat die onderdelen van het compilatieproces die afhankelijk zijn van de target taal, die onafhankelijk is van de source taal en enkel afhankelijk van de intermediaire taal [2]. Code optimalisatie en code generatie vormen samen met de gepaste foutenrapportering en de nodige operaties op de symbol table de back-end van een compiler.

Indien de back-end en de front-end van een compiler goed van elkaar gescheiden zijn is het mogelijk verschillende back-ends aan één front-end te koppelen en zo verschillende compilers te bekomen voor dezelfde source taal maar met verschillende target talen.
Een andere mogelijkheid is dezelfde back-end te gebruiken voor verschillende source talen. Elke source taal wordt vervolgens gecompileerd naar dezelfde intermediaire taal die met de gemeenschappelijke compiler back-end wordt verwerkt. Op die manier worden verschillende compilers voor dezelfde machine gecreëerd. De moeilijkheid situeert zich hier in het definiëren van een intermediaire taal. Deze taal dient namelijk de verschillende concepten uit de verschillende source talen te ondersteunen, wat niet eenvoudig is.

1.4 Front-end

Front-end : De *front-end* van een compiler bestaat uit de fasen die hoofdzakelijk afhankelijk zijn van de source taal en onafhankelijk van de target machine [2].

De onderdelen lexicale analyse, grammaticale analyse, semantische analyse en intermediaire code generatie behoren tot deze component.



Figuur 1.3: Communicatie lexicale en grammaticale analyse [2]

1.4.1 Lexicale analyse

Lexicale analyse : De hoofdtaak van de *lexicale anlyse* is het lezen van input karakters en het produceren van een sequentie van tokens die de parser gebruikt voor de grammaticale analyse [2].

Parsen : *Parsen* is de combinatie van de lexicale en de grammaticale analyse (zie 1.4.3).

De interactie tussen de lexicale analyse en de grammaticale analyse is in het algemeen van die aard dat de grammaticale analyse de lexicale analyse oproept.

De lexicale analyse wordt gescheiden van de grammaticale analyse om verschillende redenen.
Ten eerste wordt het ontwerp van de compiler eenvoudiger als beide apart gehouden worden. Het ontwikkelen van de deelcomponenten is eenvoudiger dan het ontwerpen van het geheel (cfr. verdeel en heers).
Ten tweede wordt de efficiëntie van de compiler vergroot door speciale buffertechnieken te gebruiken voor het lezen van de source code. Bij het combineren van de lexicale en grammaticale analyse zijn dergelijke technieken minder toepasbaar.
Ten derde kunnen bepaalde bijzonderheden van een input alfabet door de lexicale analyse verborgen worden waardoor portabiliteit vergroot.

Drie voorname begrippen in de lexicale analyse zijn *patroon*, *token* en *lexeem*.

Patroon : Een *patroon* is een definitie of beschrijving van de mogelijke lexemen in het source programma die een token representeren.

Token	Voorbeeld lexeems	Beschrijving patroon
const	const	constante
relatie	+, -, *	+ of - of *
id	MSL4, Python	een letter gevolgd door letters en cijfers
int	3, -5	een geheel getal

Tabel 1.1: Voorbeelden tokens

Token : Een *token* is de waarde die door de lexicale analyse aan een patroon wordt gehecht.

Lexeem : Een *lexeem* is een sequentie van karakters in het source programma dat correspondeert met een patroon van een token [2].

Nu duidelijk is waarvoor een token staat, is het nodig de verschillende tokens te kunnen definiëren. Hiervoor bestaan twee methodes die elkaar aanvullen:

- Reguliere expressie
- Reguliere definitie

Vooraleer een beschrijving van deze twee begrippen wordt gegeven, dienen eerst een aantal andere termen te worden uitgelegd:

Alfabet : Een *alfabet* is een eindige verzameling van symbolen [2].

String : Een *string*, afgeleid van een alfabet, is een eindige sequentie van symbolen uit dat alfabet [2].

Lege string : Een *lege string* (notatie ϵ) is een string met lengte nul [2].

Taal : Een taal is om het even welke verzameling van strings afgeleid uit een vast alfabet [2].

De begrippen patroon, token en lexeem worden aan de hand van voorbeelden in tabel 1.1 verduidelijkt.
Alle tokens zullen in het verdere verloop van deze thesis in het vet gedrukt worden.

Een reguliere expressie wordt opgebouwd door middel van definitieregels uit andere (eenvoudigere) reguliere expressies. De definitie van een reguliere expressie is bijgevolg een recursieve definitie.
Een reguliere expressie s bepaalt de taal $L(s)$.

Reguliere expressie : Reguliere expressies over een alfabet Σ worden bepaald door de regels [2]:

1. ϵ is een reguliere expressie die de taal $\{\epsilon\}$ definieert.
2. Indien een symbool a een element is van Σ dan is a een reguliere expressie die $\{a\}$ bepaalt.
3. Stel dat r en s reguliere expressies zijn die respectievelijk de talen $L(r)$ en $L(s)$ bepalen, dan is:
   (a) $(r)|(s)$ een reguliere expressie die de taal $L(r) \bigcup L(s)$ bepaalt.
   (b) $(r)(s)$ een reguliere expressie die de taal $L(r)L(s)$ bepaalt.
   (c) $(r)*$ een reguliere expressie die de taal $L(r)*$ bepaalt.
   (d) (r) een reguliere expressie die de taal $L(r)$ bepaalt.

In de definitie staat | voor "of", is $(r)(s)$ de concatenatie van r en s en heeft $(r)*$ de betekenis "nul of meer concatenaties van r".

Voorbeeld : Een voorbeeld van een reguliere expressie die een natuurlijk getal definieert, met **cijfer** het token dat een cijfer voorstelt:

$$\mathbf{cijfer}(\mathbf{cijfer})^*$$

Een naam geven aan een reguliere expressie maakt het mogelijk die reguliere expressie als symbool te gebruiken in een reguliere expressie. Een manier om een naam te geven aan een reguliere expressie is het gebruik van een *reguliere definitie*.

Reguliere definitie [2] : Stel Σ een alfabet van basissymbolen, dan is een *reguliere definitie* een opeenvolging van definities van de vorm

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_i \rightarrow r_i$$

met elke d_i een unieke naam en elke r_i een reguliere expressie over de symbolen in $\Sigma \bigcup \{d_1, d_2, \ldots, d_{i-1}\}$ (de basissymbolen en de voordien gedefinieerde symbolen).

Voorbeeld : Een voorbeeld van een reguliere definitie is de reguliere definitie die de verzameling van de natuurlijke getallen voorstelt:

$$\mathbf{cijfer} \rightarrow 0 \mid 1 \mid \ldots \mid 9$$
$$\mathbf{getal} \rightarrow \mathbf{cijfer}(\mathbf{cijfer})^*$$

De herkenning van tokens in het source programma gebeurt door middel van eindige automaten, meer bepaald door *nondeterministische* en/of *deterministische eindige automaten*. Beiden bezitten een starttoestand en één of meerdere eindtoestanden. Door het lezen van input karakters uit het source programma gaat de automaat over van de ene naar de andere toestand om uiteindelijk al of niet in een eindtoestand terecht te komen. Een input string wordt aanvaard indien bij het lezen finaal een eindtoestand bereikt wordt. De lexicale analyse tracht altijd een zo lang mogelijke string te matchen met een patroon en daarvoor een token te creëren.
Voor een uitgebreidere theorie over de toepassing van deze automaten wordt verwezen naar [2] en [25].

Tijdens een lexicale analyse kunnen fouten optreden indien een input string geen enkel van de gedefinieerde patronen volgt. Hierop kan op verschillende manieren gereageerd worden:

- de lexicale analyse geeft het onmiddellijk op en meldt dat het source programma niet voldoet aan de lexicale voorwaarden.
- input karakters worden verwijderd tot er wel aan een gespecificeerd patroon voldaan wordt.
- er wordt een mogelijk ontbrekend karakter toegevoegd.
- een fout karakter wordt vervangen door een juist karakter.
- twee opeenvolgende karakters worden van plaats verwisseld.

Uitgezonderd de eerste manier van omgaan met fouten, trachten de overige vier de input te verbeteren. De meest eenvoudige manier om een fout source programma te verbeteren is door slechts één karakter te veranderen. Er zijn eveneens technieken die het minimum aantal correcties berekent die nodig zijn om een juiste input te hebben . Deze technieken worden echter door hun complexiteit en rekentijd weinig gebruikt [2].

De lexicale analyse zorgt tevens voor het strippen van het source programma: commentaar en witruimte worden verwijderd uit de code. Dit is op een eenvoudige manier mogelijk doordat er een scheiding is met de grammaticale analyse.

De theoretische gedachte achter een lexicale analyse wordt door verschillende computer tools geïmplementeerd. Hierna wordt nader ingegaan op de meest gebruikte tool, met name *lex* (lexical analyzer generator) waar ondertussen een betere versie van beschikbaar is, namelijk *flex* [6] (fast lexical analyzer generator). Flex is vrij beschikbaar en wordt via het GNU-project [30] verspreid.

Lex - flex

Een lexicale specificatie kan gebeuren in lex. Op basis van een lexicale specificatie genereert lex C-code die de lexicale specificatie implementeert.

Een lex-specificatie is opgebouwd uit drie delen [6]: de *definitiesectie*, de *regelsectie* en de *gebruiker subroutines sectie*. De delen worden gescheiden door een regel waarop twee procent symbolen staan.

```
...definitiesectie...
%%
...regelsectie...
%%
...gebruiker subroutines sectie...
```

Definitiesectie : Een *definitiesectie* bevat de C/C++-code die we letterlijk gecopieerd willen zien in het uiteindelijke programma. Hier worden onder andere header bestanden geïncludeerd. De C/C++-code wordt omgeven door "%{" en "%}". In de sectie komen ook de *definities* te staan. Het einde van de sectie wordt met "%%" aangegeven.

Definitie : Een *definitie* biedt de mogelijkheid lange of complexe *patronen* te vervangen door een identifier (zie verder voor een voorbeeld).

Regelsectie : De regels in de *regelsectie* bestaan uit een *patroon* en een *actie* die gescheiden worden door witruimte. De lexer (zo wordt het uiteindelijke programma genoemd) voert voor elk gevonden patroon de respectievelijke actie uit. Het einde van de sectie wordt zoals de vorige gemarkeerd met "%%".

Patroon : Een *patroon* is een opeenvolging van karakters die voldoen aan een bepaalde definitie of beschrijving.

Actie : Een *actie* is de programmacode die uitgevoerd wordt wanneer een patroon herkend wordt.

Gebruiker subroutines sectie : Een *gebruiker subroutines sectie* betstaat uit C/C++-code die na de door lex gegenereerde code wordt geplaatst.

Lex produceert uit deze verschillende secties een routine (de lexer) **yylex()** die instaat voor de lexicale analyse van de input. De lexer overloopt zijn invoerkarakters van links naar rechts.

Het volgende voorbeeld hanteert een lexicale definitie die witruimte, integer getallen, +, * en - herkent als patronen. Het lijnnummer wordt bijgehouden zodat de lijn, waar eventueel een fout in de input staat, kan gemeld worden aan de gebruiker. De mogelijkheid bestaat om commentaar in het source programma op te nemen. De lexicale specificatie in flex-vorm is dan:

```
%{

#include <stdio.h>
#include "y.tab.h"

extern int lineno;

%}

ws [ \t\r]+
comment "--".*
nl \n
number [0-9]+

%%

{ws} ;

{comment} {fprintf(stdout, "A line of comment was parsed.\n");}

{number} { yylval.number = atoi(yytext); return NUMBER; }
```

```
"+"         { return PLUS; }

"*"         { return MUL; }

"-"         { return INV; }

{nl}        { lineno++; }

.           { return yytext[0]; }

%%

void
yyerror(char *msg)
{
  fprintf(stderr, "%d: %s at '%s'\n", lineno, msg, yytext);
}
```

De drie onderdelen van een lex-specificatie zijn aanwezig in de code. De meeste acties in de regelsectie zijn van het type "`return`". Dergelijke regels geven een token terug aan het programma dat de functie yylex() oproept. Dit zal in de meeste gevallen de grammaticale analyse zijn. Een token wordt door de lexer aan de parser doorgegeven. In de parser corresponderen de tokens met terminals in de productieregels.

In de code komen ook een aantal definities voor. Bijvoorbeeld `ws [ \t\r]+`: deze definitie bepaalt dat "ws"[1] bestaat uit één of meer instanties van een spatie, een tabulator of een carriage return.

1.4.2 Symbol table

Een symbol table is een belangrijk onderdeel van een compiler om informatie over de verschillende attributen van de identifiers in het source programma te verzamelen. Het gaat dan om het type, de scope, het aantal argumenten, het type van de argumenten, ...

Symbol table : Een *symbol table* is een datastructuur die een record bevat voor elke identifier van het source programma, met velden voor de attributen van die identifier [2].

[1] ws is de afkorting voor white space of in het Nederlands witruimte.

Als een identifier door de lexicale analyse herkend wordt in het source programma, wordt de identifier toegevoegd aan de symbol table indien hij nog niet aanwezig is. Een symbol table moet dus toelaten om efficiënt informatie toe te voegen en op te vragen. Tijdens de lexicale analyse kunnen niet alle attribuutwaarden van een identifier bepaald worden. De fasen volgend op de lexicale analyse voegen informatie toe aan de symbol table en maken gebruik van de reeds aanwezige informatie. Bij semantische analyse en intermediaire code generatie is het bijvoorbeeld nodig het type van de identifiers te kennen om na te gaan of er geen fouten in het source programma staan.

Als datastructuur voor de symbol table kan gebruik gemaakt worden van een array, een lijst of een hashtabel. Een lijst is eenvoudig te implementeren maar de toegang is trager bij een groot aantal elementen en bevragingen. Hashing is efficiënter maar is moeilijker te implementeren.
Een symbol table kan dynamisch groeien tijdens de loop van een compilatie of kan een vooraf bepaalde grootte hebben. Een symbol table met een vooraf bepaalde grootte moet voldoende groot gekozen worden om elk source programma te kunnen verwerken. Een symbol table met een vaste grootte zal voor de meeste source programma's te groot zijn en voor enkele te klein. Dergelijke manier van werken is niet de meest geschikte.

Elk element van een symbol table hoeft niet van dezelfde vorm te zijn. Voor elke identifier dient immers niet dezelfde informatie bewaard te worden.

1.4.3 Grammaticale analyse - Parser

Het onderscheid tussen lexicale en grammaticale analyse is niet altijd even duidelijk. Bepaalde taalconstructies kunnen bijvoorbeeld ook in de lexicale specificaties worden opgenomen. Recursie is een constructie die niet in de lexicale analyse thuishoort. Lexicale constructies hebben geen behoefte aan recursie, syntax constructie vaak wel.

Een grammatica van een progammeertaal geeft op een verstaanbare manier weer hoe de taal syntactisch is opgebouwd. Uitgaande van de grammatica kan bepaald worden of een source programma syntactisch correct is.

Parser : Een *parser* krijgt een string van tokens van de lexicale analyse (zie figuur 1.4) en verifieert of deze string voldoet aan de grammatica van de source taal.

In de voorgaande definite is een *string van tokens* een sequentie van nul of meer tokens. De representatie van de *lege string* is ϵ. De string van tokens die een parser ontvangt is afkomstig van de lexicale analyse.

Er worden drie types parsers onderscheiden [2]:

- **Universele parsing methodes** : Een voorbeeld van een universele parsing methode is het Cocke-Younger-Kasami algoritme. Deze methodes zijn niet efficiënt genoeg om in de praktijk te worden gebruikt.
- **Top-down parsers** : Deze parsers bouwen de abstract syntax tree op beginnend met de wortel en zo naar de bladeren toe.

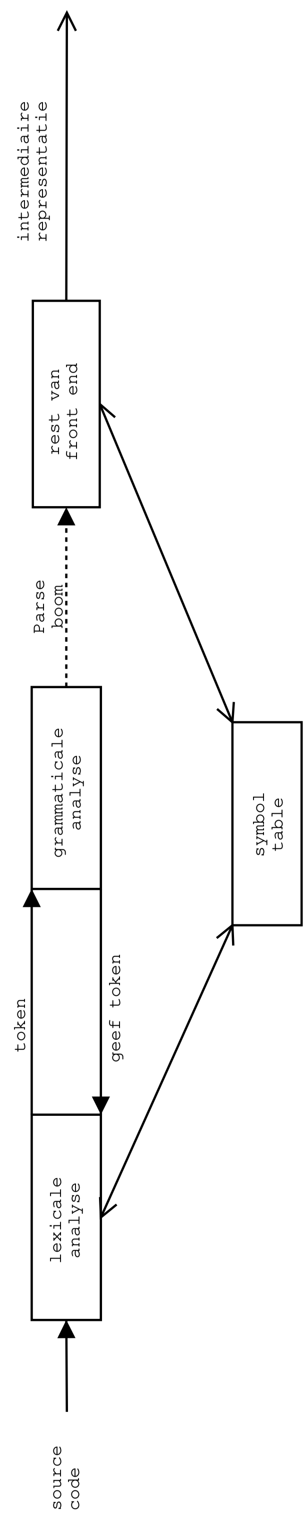
  Abstract Syntax Tree : Een *abstract syntax tree* is een boomvoorstelling van het source programma (zie pagina 17).
- **Bottom-up parsers** : deze parsers creëren eerst de bladeren en bouwen van daaruit de abstract syntax tree op.

De input van de parser bij top-down en bottom-up parsing wordt van links naar rechts verwerkt.

Contextvrije grammatica

Contextvrije grammatica's worden gebruikt om de syntax van een taal te beschrijven. Ze bestaan uit vier componenten:

1. Een verzameling *terminale symbolen* (basissymbolen waaruit strings worden opgbouwd) of tokens.
2. Een verzameling *niet-terminale symbolen* (syntactische variabelen die een string bepalen).
3. Een verzameling van *producties*. Een productie bestaat uit een niet-terminaal symbool, het linkerlid, een pijl en een rij van terminale en/of niet-terminale symbolen, het rechterlid van de productie.
4. Een niet-terminaal symbool dat dienst doet als *startsymbool*. Van het startsymbool kan de volledige taal worden afgeleid.



Figuur 1.4: Parser in compilermodel [2]

Niet-terminalen worden in de loop van deze thesis *schuin* afgebeeld.
De conventie is dat de grammatica aanvangt met de productie van het startsymbool. Deze conventie zal hierna gevolgd worden.

Strings worden van een grammatica afgeleid vertrekkend van het startsymbool en daarna het herhaaldelijk vervangen van niet-terminalen door het rechterlid van een productie waarvan dit niet-terminaal symbool het linkerlid is. De strings van tokens die kunnen afgeleid worden van het startsymbool vormen de taal gedefinieerd door de grammatica.

Voorbeeld : Een voorbeeld van een contextvrije grammatica specificeert expressies opgebouwd uit cijfers en de symbolen + en -. Een + of - staat steeds tussen twee cijfers.

$$\begin{aligned} lijst &\rightarrow lijst \mathbf{+} lijst \\ lijst &\rightarrow lijst \mathbf{-} lijst \\ lijst &\rightarrow cijfer \\ cijfer &\rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \mid \end{aligned}$$

De laatste productie is de gegroepeerde productie van de tien producties van de vorm

$$cijfer \rightarrow \mathbf{i} \text{ met } \mathbf{i} = \mathbf{1}, \ldots, \mathbf{10}.$$

Het teken "|" heeft ook hier de betekenis van "of". De tokens van deze grammatica zijn:

+ - 0 1 2 3 4 5 6 7 8 9

Contextvrije grammatica's hebben enkel nut als men wil controleren op de syntactische correctheid van het source programma. Voor praktisch gebruik van een compiler is het nodig dat het source programma tijdens de verwerking door de parser wordt vertaald naar een andere representatie. Syntax gestuurde vertaling is daarvoor de gepaste oplossing.

Abstract Syntax Tree - AST

Door het gebruik van een contextvrije grammatica kan het source programma gerepresenteerd worden met behulp van een boom in plaats van tekst.

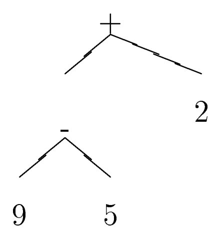
Abstract Syntax Tree : Een *abstract syntax tree* is een boom die een correct source programma voorstelt.

De boom voldoet aan de volgende eigenschappen:

1. De wortel van de boom heeft het startsymbool als label.
2. Elk blad heeft een token of ϵ als label.
3. Elke knoop draagt als label een niet-terminaal symbool.
4. Als A het label is van een knoop met als kinderen X_1, X_2, …, X_n, dan is $A \rightarrow X_1$ X_2 … X_n een productie. Hierin zijn X_1, X_2, …, X_n terminale of niet-terminale symbolen. Een speciaal geval is de productie $A \rightarrow \epsilon$ die een knoop met één kind met label ϵ voorstelt.

Een abstract syntax tree geeft de natuurlijke hiërarchische structuur van een source programma weer.

Voorbeeld : De volgende abstract syntax tree is de voorstelling van de expressie 9-5+2 bepaald door de voorbeeldgrammatica besproken in 1.4.3.



Aan elk van de knopen en bladeren van de abstract syntax tree kunnen semantische acties gekoppeld worden. De twee bestaande technieken zijn opgenomen in de volgende sectie.

Syntax gestuurde vertaling (Syntax-directed translation)

Om een source taal te vertalen dient de compiler met meer dan alleen de sequentie van de tokens rekening te houden. Aan bepaalde constructies (terminalen, niet-terminalen, …) in de grammatica worden attributen of acties verbonden. De attributen krijgen tijdens het parsen een waarde toegekend en de acties worden uitgevoerd.

Er zijn twee manieren om een syntax gestuurde vertaling te bereiken:

1. **Syntax gestuurde definitie** : Een *syntax gestuurde definitie* beschrijft de vertaling van een constructie door middel van de attributen die geassocieerd zijn met de syntactische componenten [2]. Op de attributen worden semantische acties gedefinieerd.

2. **Vertalingssysteem** : Een *vertalingssysteem* is een contextvrije grammatica waar attributen verbonden worden met grammaticasymbolen en in de rechterleden van de producties *semantische acties* voorkomen [2].

Voorbeeld : Een voorbeeld van een syntax gestuurde definitie is een productie die de som van een expressie (*expr*) en een term (*term*) definieert. De waarde wordt in het attribuut t van de expressie (*expr*) en de term (*term*) bewaard.

Productie	Actie
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.\mathrm{t} + term.\mathrm{t}$

Deze manier van werken wordt uitgebreid besproken in [2]. Er wordt in deze thesis niet verder op ingegaan.

Een vertalingssysteem is vergelijkbaar met een syntax gestuurde definitie met dat verschil dat de volgorde van evaluatie van de semantische acties bepaald is. De plaats waar een semantische actie dient uitgevoerd te worden wordt bepaald door de plaats waar de actie staat in het rechterlid van een productie. De semantische acties worden omgeven door accolades (zie het volgende voorbeeld).

Een vertalingsschema vertaalt een source programma door de acties, die geassocieerd zijn met een productie, uit te voeren op het moment dat de productie wordt gebruikt.

Voorbeeld :

$$
\begin{array}{ll}
expr \rightarrow expr + term & \{ \text{print('+')} \} \\
expr \rightarrow expr \text{ - } term & \{ \text{print('-')} \} \\
expr \rightarrow term & \\
term \rightarrow 0 & \{ \text{print('0')} \} \\
term \rightarrow 1 & \{ \text{print('1')} \} \\
\ldots & \\
term \rightarrow 9 & \{ \text{print('9')} \}
\end{array}
$$

De acties in dit voorbeeld zijn enkel `print` commando's. Andere acties zoals de creatie van nieuwe variabelen en objecten zijn ook mogelijk en zorgen voor de opbouw van een nieuwe representatie van het source programma (zie verder).

De manier waarop acties aan producties worden gekoppeld is nu beschreven. Door gepaste acties in de definitie op te nemen wordt een abstract syntax tree opgebouwd tijdens het parsen.

Soorten Parsers

Niet alle parsers construeren op dezelfde manier de abstract syntax tree. Afhankelijk van de manier waarop ze dit doen kunnen bepaalde grammatica's al of niet verwerkt worden. Zoals reeds eerder vermeld wordt een opdeling gemaakt in drie groepen. Twee groepen zijn in de praktijk bruikbaar: de top-down en de bottom-up parsers.

- Top-down parser
  - Recursive-Descent Parsing
  - Predictive Parser
  - Nonrecursive Predictive Parsing
- Bottom-up parser
  - Shift-reduce parsing
    - Operator-precedence parsing
    - LR(k) parsing

Hierna volgt een bespreking van de LR parsing methode omdat deze methode gebruikt wordt door *bison*, de tool die in het kader van deze thesis aangewend wordt om een parser te implementeren. Voor de andere parser types wordt verwezen naar [2].

LR(k) parsing

Een LR(k) parser is een shift-reduce parser, wat er op neerkomt dat hij instaat voor een bottom-up syntax analyse. De "L" staat voor het feit dat de input van links naar rechts wordt gelezen. De "R" wil zeggen dat de meest rechtse afleiding zal geselecteerd worden

$$
\begin{aligned}
START &\rightarrow expression && (1.1)\\
expression &\rightarrow expression\, PLUS\, term && (1.2)\\
&\mid term && (1.3)\\
term &\rightarrow term\, MUL\, factor && (1.4)\\
&\mid factor && (1.5)\\
factor &\rightarrow \text{'('}\, expression\, \text{')'} && (1.6)\\
&\mid INV\, number && (1.7)\\
&\mid number && (1.8)\\
number &\rightarrow NUMBER && (1.9)
\end{aligned}
$$

Figuur 1.5: Contextvrije grammatica

in een productie en "k" staat voor het aantal *lookahead symbolen*. Indien "(k)" niet wordt vermeld, is k gelijk aan 1 ondersteld en wordt gesproken van een LR parser.

LR(k) parsing heeft verschillende voordelen:

- het kan gebruikt worden voor bijna elke taal waarvoor een contextvrije grammatica kan gespecificeerd worden.
- het is een algoritme dat niet aan backtracking doet maar toch efficiënt kan geïmplementeerd worden.
- het kan een syntactische fout ontdekken zodra het mogelijk is bij het overlopen van de input van links naar rechts.

Het grote nadeel is dat LR(k) parsers veel werk vragen om ze handmatig te implementeren. Er zijn echter verschillende parser generators te vinden die dit werk automatiseren. Zo een generator zet een contextvrije grammatica om naar een LR(k) parser voor de grammatica.

Nu wordt de werking van een LR(1) parser op een intuïtieve manier van naderbij bekeken. Voor een grondige theoretische uitleg dient verwezen te worden naar [2].

De gehanteerde grammatica is opgenomen in figuur 1.5.

De gebruikte parser is een *shift-reduce parser*. Dit wil zeggen dat de parser een productie van rechts naar links zal uitvoeren indien mogelijk (*reduce*) (rekening houdend met de resulterende toestand) en zoniet een nieuw token zal vragen aan de lexer (*shift*).

De parser maakt gebruik van één "lookahead" symbool. Dit komt er op neer dat de parser één token verder kijkt om te bepalen of een productie al dan niet kan toegepast worden. De productie met het hoogste niveau is standaard de eerste productie. Dit geeft aan dat de volledige invoerstring dient te voldoen aan deze productieregel.

Het gehanteerde voorbeeld is de expressie 1+2*3. Figuur 1.6 toont de evolutie van het parsen.
De cijfercode boven de pijltjes verwijst naar de toegepaste productie uit figuur 1.5. De schuingedrukte tekst is het lookahead symbool.
Bij de overgang van de toestand 1.6(a) naar de toestand 1.6(b) wordt de productie 1.9 toegepast. Het token `NUMBER` wordt gesubstitueerd door het niet-terminaal symbool `number`.
Bij overgang van 1.6(j) naar 1.6(k) wordt de productie 1.4 toegepast en niet 1.2. Een reductie aan de hand van de productieregel 1.2 zou immers leiden tot de volgende toestand:

MUL
`expression`

Deze situatie kan nooit door één van de voorhanden zijnde regels gereduceerd worden. Bijgevolg zal de reductie van 1.6(j) met behulp van 1.2 niet worden uitgevoerd.

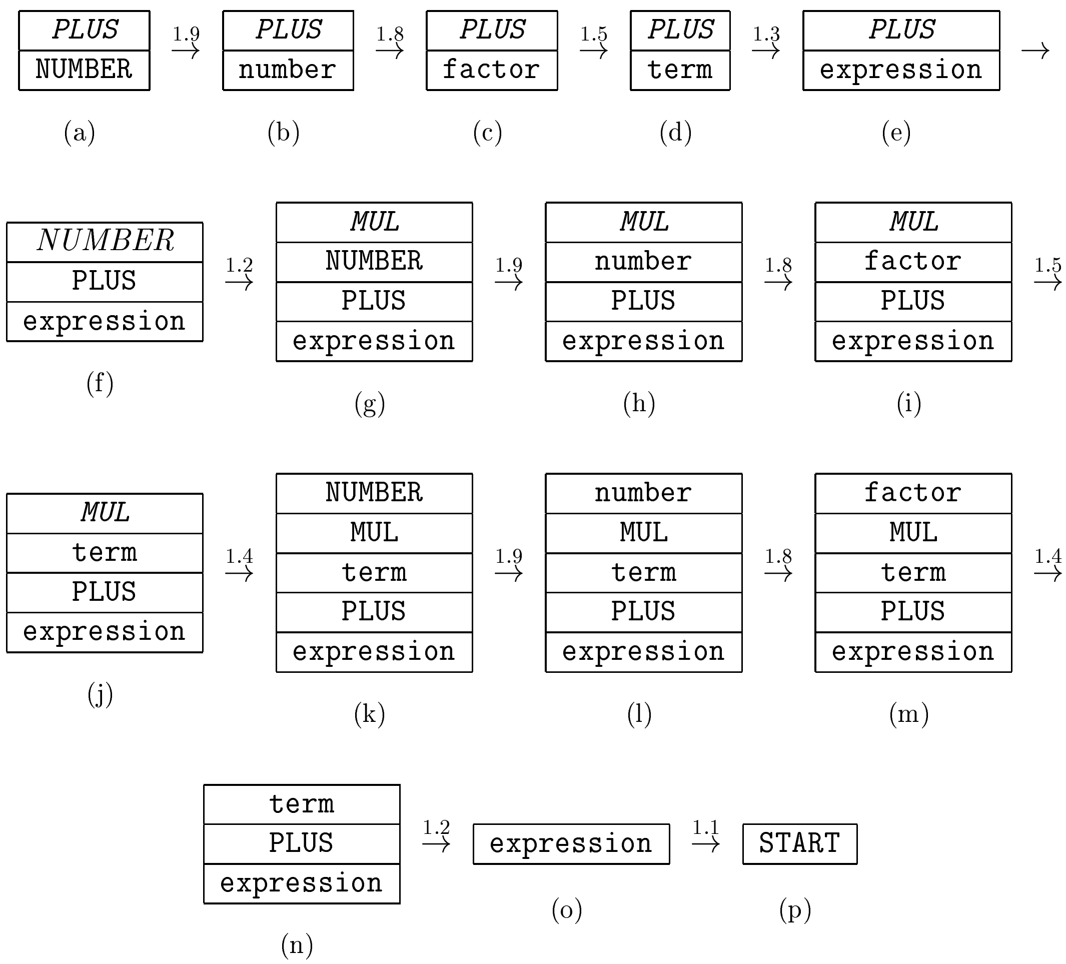
Yacc - Bison

De tool, yacc of bison, die zal gebruikt worden in hoofdstuk IV, is een LR(1) parser.

Yacc (Yet anothor compiler compiler) en bison zijn twee programma's die gebruikt worden om een parser automatisch te genereren [6] [3]. Bison is de verbeterde GNU [30] versie van yacc. De twee tools krijgen als input een grammatica en geven als output een parser in de vorm van C-code. Er zijn verschillen in gebruik van yacc en bison. De hierna volgende bespreking handelt over bison.

Bison produceert uitgaande van de gespecificeerde grammatica een LR(1) parser. De structuur van een bison file is analoog aan die van een flex file:

```
...definitiesectie...
%%
...regelsectie...
```



Figuur 1.6: Parse voorbeeld 1+2*3

```
%%
...gebruiker subroutines sectie...
```

De definitiesectie bevat eventueel C-code die letterlijk wordt overgenomen in de parser code. Hier kunnen ook declaraties van de vorm `%start` (geeft aan wat het startsymbool is), `%type` (het type van een symbool) en andere parser informatie vermeld staan.

In de regelsectie worden de grammaticale producties en acties gespecificeerd. Een bison regelsectie specificatie is eigenlijk een grammatica in de vorm van een vertalingsschema.

Voorbeeld : Het voorbeeld op pagina 19 wordt vertaald naar bison code, voor de regelsectie, tot:

```
expr : expr '+' term { fprintf(stdout,"+"); };
expr : expr '-' term { fprintf(stdout,"-"); };
expr : term;
| '0'{ fprintf(stdout,"0"); }
| '1'{ fprintf(stdout,"1"); }
| '2'{ fprintf(stdout,"2"); }
| '3'{ fprintf(stdout,"3"); }
| '4'{ fprintf(stdout,"4"); }
| '5'{ fprintf(stdout,"5"); }
| '6'{ fprintf(stdout,"6"); }
| '7'{ fprintf(stdout,"7"); }
| '8'{ fprintf(stdout,"8"); }
| '9'{ fprintf(stdout,"9"); };
```

In de gebruiker subroutines sectie kunnen zoals in een flex-bestand (zie pagina 11) zelfgedefinieerde subroutines geplaatst worden die overgenomen worden in de geproduceerde C-code.

1.4.4 Semantische analyse

Een source programma kan lexicaal en syntactisch volledig aan de gestelde voorwaarden voldoen maar toch is het mogelijk dat het geen juiste semantische betekenis heeft. Er kan bijvoorbeeld een integer waarde aan een instantie van een zelfgedefinieerde klasse worden toegekend. Controle op de semantische juistheid van de aangeboden code is dan ook nodig.

Semantische analyse : Een *semantische analyse* voert controles uit op het semantisch correct zijn van het source programma en verzamelt type-informatie voor de volgende fase, met name de code generatie [2].

Een belangrijk onderdeel van de semantische analyse is type checking.

Type checking : *Type checking* gaat na of het type van de operanden van een operator van het type zijn die in de grammatica gespecificeerd zijn.

Veel programmeertalen geven bijvoorbeeld een fout indien getracht wordt een rij te indexeren met een reëel getal. Type checking wordt niet altijd strikt toegepast. Het toekenen van een integer waarde aan een reële variabele is meestal wel toegelaten, maar er gaat dan wel een conversie van een reële naar een gehele waarde aan vooraf.

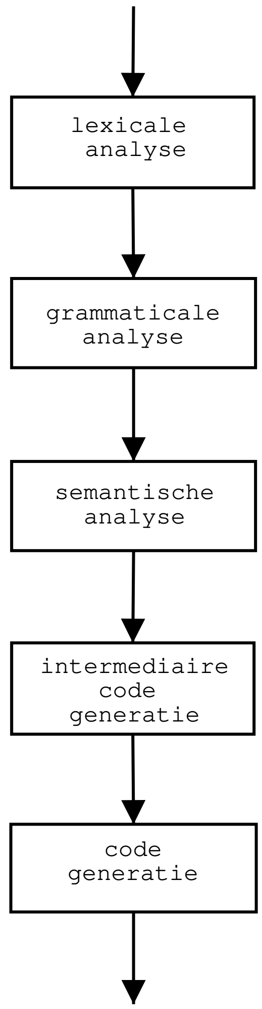
Een andere semantische controle is de *flow-of-control check*. Code die er voor zorgt dat een bepaalde constructie verlaten wordt, moet naar een plaats in de code gaan waar het programma kan verdergezet worden. Zo verlaat bijvoorbeeld een `break` statement, in C, de dichtste `while` lus of `switch` statement. Er dient dan wel een omsluitende `while` of `switch` te zijn, zoniet treedt er een fout op.

Tevens kan er een controle uitgevoerd worden op het *uniek* zijn van sommige objecten. Er zijn situaties waar het nodig is dat een object uniek is. In Pascal dient een identifier bijvoorbeeld uniek gedefinieerd te zijn en labels van een *case* statement dienen eveneens uniek te zijn.
De controle op het uniek zijn kan gebeuren als een nieuwe naam wordt toegevoegd aan de symbol table.

Naamgerelateerde controles kunnen eveneens nodig zijn. Een naam moet soms twee of meerdere malen voorkomen in een bepaalde constructie. In Ada dient bij het begin en het einde van een `loop`-blok dezelfde naam gebruikt te worden.

1.4.5 Intermediaire code generatie

Intermediaire code generatie wordt niet altijd uitgevoerd aangezien het perfect mogelijk is een source programma onmiddellijk te vertalen naar zijn overeenkomstig target programma.



Figuur 1.7: Positie intermediaire code generatie

Het mogelijk gebruik van intermediaire code generatie is het vertalen van verschillende source talen, door een eigen front-end, naar een intermediaire taal. De gegenereerde code kan door een zelfde back-end verwerkt worden. De tussentijdse code moet aan twee belangrijke voorwaarden voldoen:

1. ze moet eenvoudig te produceren zijn.

2. ze moet eenvoudig vertaalbaar zijn naar de target taal.

De plaats die deze fase in het model inneemt is te zien in figuur 1.7.

De voordelen bij het gebruik van een intermediaire code zijn:

- De overdraagbaarheid wordt vereenvoudigd. Een compiler voor een andere machine kan gebouwd worden door een back-end van de andere machine te koppelen aan een bestaande front-end.
- De intermediaire code kan door een machine onafhankelijke code optimalisator geoptimaliseerd worden.

Het is mogelijk de intermediaire code generatie te integreren met de parsing fase van het compilatieproces.
Net als een abstract syntax tree is een intermediaire code een tussentijdse tekstuele representatie van een source programma. Als algemene intermediaire code wordt *drie adres code* beschouwd.

Drie adres code

Drie adres code : *Drie adres code* is een sequentie van statements met als algemene vorm:

$$\mathtt{x} := \mathtt{y}\ op\ \mathtt{z}$$

met `x`, `y` en `z` namen, constanten of tijdelijke compilersymbolen en *op* een operator, bijvoorbeeld een wiskundige bewerking op integer waarden of een logische bewerking op boolean variabelen [2].

Zo wordt `a := x * y + z` mogelijk vertaald in de volgende drie adres code sequentie:

$$\begin{aligned} \mathtt{t}_1 &:= \mathtt{x} * \mathtt{y} \\ \mathtt{t}_2 &:= \mathtt{t}_1 + \mathtt{z} \\ \mathtt{a} &:= \mathtt{t}_2 \end{aligned}$$

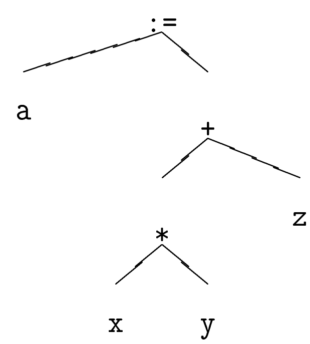
Met $\mathtt{t}_1$ en $\mathtt{t}_2$ door de compiler gedefinieerde tijdelijke namen.

Drie adres code voldoet aan de volgende eigenschappen:

- buiten de assignatie is er ten hoogste één operator.
- de compiler gebruikt tijdelijke variabelen om tussentijdse resultaten op te slaan.
- bepaalde drie adres instructies hebben minder dan drie operanden (b.v. `a := temp1`).

Wiskundige expressies en control statements worden herleid naar een aantal drie adres code expressies. Dit maakt, door de eenvoudigere optimalisatie, drie adres code aantrekkelijk als intermediaire code. Door het gebruik van tijdelijke waarden kunnen statements makkelijk verplaatst worden.

Drie adres code is de lineaire representatie van een abstract syntax tree met als knopen de operatoren van de statements en als bladeren de niet-tijdelijke variabelen. De voorstelling als abstract syntax tree van `x * y + z` is:



De knoop met label `+` stemt overeen met $\texttt{t}_2$ en die met label `*` met $\texttt{t}_2$.

Om drie adres code te bekomen dient een syntax gestuurde vertaling geïmplementeerd te worden die, uitgaande van de source code, drie adres code zal produceren.

1.5 Back-end

De back-end van een compiler bestaat uit de fasen die afhankelijk zijn van de target taal. De back-end bestaat uit twee onderdelen: code optimalisatie en code generatie. Code generatie wordt, in tegenstelling tot code optimalisatie, altijd uitgevoerd.

1.5.1 Code optimalistie

In het ideale geval produceren compilers code die van dezelfde kwaliteit is als wat kan bereikt worden met handgeschreven code. Dit doel wordt slechts in enkele gevallen met moeite bereikt. Het is vaak eenvoudig om door een compiler geproduceerde code handmatig te optimaliseren naar tijd en ruimte. Optimalisatie van de code kan dit probleem verkleinen. Code optimalistie wil niet zeggen dat optimale code zal bekomen worden maar wel dat zal getracht zal worden een zo goed mogelijk resultaat te bekomen.

Optimaliserende compilers : *Optimaliserende compilers* zijn compilers die codeverbeterende transformaties uitvoeren.

De nadruk bij het optimaliseren wordt vooral gelegd op het verbeteren van die delen van een programma die het meest worden uitgevoerd. Door een kleine verbetering van de code kan dan een sterke reductie van de nodige bronnen behaald worden. Meestal is het namelijk het geval dat slechts een klein deel van de code verantwoordelijk is voor het grootste deel van de uitvoeringstijd. Het spreekt voor zich dat lussen een grote belangstelling zullen genieten van een optimalisator.

Er bestaan verschillende criteria voor de selectie van mogelijke optimalisaties:

1. Een tranformatie moet de betekenis van een programma behouden. Een optimalisatie mag niet de output van een progamma veranderen of een fout veroorzaken die niet aanwezig was in het originele programma.
2. Een transformatie moet een programma meetbaar versnellen.
3. Een transformatie moet de moeite waard zijn. Het is niet nuttig dat de tijd die nodig is om te transformeren langer is dan de tijd die gewonnen wordt bij het uitvoeren van het target programma. Het is ook niet nodig veel energie te stoppen in een optimalisatie van een programma dat slechts enkele malen zal worden uitgevoerd.

Een mogelijke optimalisatie van de drie adres code sequentie

$$\begin{aligned} \mathtt{t}_1 &:= \mathtt{x} * \mathtt{y} \\ \mathtt{t}_2 &:= \mathtt{t}_1 + \mathtt{z} \\ \mathtt{a} &:= \mathtt{t}_2 \end{aligned}$$

is de volgende:

$$\begin{aligned} \mathtt{t}_1 &:= \mathtt{x} * \mathtt{y} \\ \mathtt{a} &:= \mathtt{t}_1 + \mathtt{z} \end{aligned}$$

.

De looptijd van een programma kan niet enkel verbeterd worden door gebruik te maken van een code optimalisator maar kan ook op andere niveaus gebeuren, zoals op het niveau van het source programma.

Geen enkele compiler kan voor elk probleem optimale code genereren. Het produceren van optimale code is immers een onbeslisbaar probleem [2].

1.5.2 Code generatie

De laatste fase in het compilatieproces is de generatie van de target code (figuur 1.7). Meestal is deze code porteerbare machine code, assembler code of code in een andere taal dan de source taal.

Code generatie : De *code generatie* krijgt als input een intermediaire representatie van het source programma en produceert als output een equivalent target programma.

Aan de code generatie worden bepaalde eisen gesteld. De output code moet juist zijn en van goede kwaliteit.

Indien de back-end en de front-end van de compiler goed van elkaar gescheiden zijn zal (indien de lexicale analyse, de grammaticale analyse, de semantische analyse en het vertalen naar een intermediaire code correct verlopen is) de input foutenvrij zijn. De code generatie krijgt als input een intermediaire voorstelling van de front-end en zal met gebruik van de symbol table het target programma creëren. Verschillende variabelen of geheugenlocaties nodig in het target programma worden aangemaakt. De intermediaire instructies worden vertaald naar target taalinstructies. Voor de code generatie wordt meestal gebruik gemaakt van heuristieken die correcte, maar niet noodzakelijk optimale code opbouwen. De keuze van de heuristiek is belangrijk en kan een weerslag hebben op de efficiëntie van de code. Het target programma is het doel van de compiler.

1.6 Besluit

Het compilatieproces kan worden opgedeeld in zes fasen: lexicale analyse, grammaticale analyse, semantische analyse, intermediaire code generatie, code optimalisatie en code generatie. Deze fasen kunnen verder opgesplitst worden in de front-end (lexicale analyse,

grammaticale analyse, semantische analyse en intermediaire code generatie) en de back-end (code optimalisatie en code generatie). De front-end en de back-end worden zo goed mogelijk van elkaar gescheiden om de herbruikbaarheid van de verschillende componenten te verhogen. De symbol table interageert met alle fasen van het compilatieproces en hoort principieel niet thuis in de front- of back-end. De opbouw start tijdens de lexicale analyse en wordt tijdens de volgende fasen aangepast en bevraagd.
Lexicale analyse gaat na of het source programma juist gespeld is. Tijdens de grammaticale analyse wordt gecontroleerd of de compiler input voldoet aan de grammatica van de taal. De semantische analyse gaat na of wat in de source staat wel een betekenis heeft. Na deze drie fasen staat er een abstract syntax tree ter beschikking van de intermediaire code generatie of, indien er geen intermediaire code generatie is, onmiddellijk van de code generatie met eventuele code optimalisatie. De geproduceerde code is het eindproduct van de compiler.

Hoofdstuk II

AToM^3

Om niet telkens experimenten uit te voeren willen wetenschappers experimenten simuleren. Om een experiment te kunnen simuleren dient er eerst een model van het experiment gemaakt te worden. Om een dergelijk model te kunnen simuleren moet er een tool beschikbaar zijn waarin het model kan gespecificeerd en gesimuleerd worden. In AToM^3 kunnen er modellen in verschillende formalismen gespecificeerd en gesimuleerd worden.

De MSL-parser die ontwikkeld werd kan vanuit AToM^3 worden opgeroepen. Het object dat de geparste code voorstelt kan dan in AToM^3 bewerkt worden.

2.1 AToM^3

AToM^3 [27] is een tool voor multi-formalisme modelleren. AToM^3 wordt nog steeds ontwikkeld in het *Modelling, Simulation and Design Lab (MSDL)* [1], *School of Computer Science* [2] van de *McGill University* [3] in Montreal, Canada.

Als men een model wil maken van een *systeem* bestaat dit in het algemeen uit een aantal componenten. Complexe systemen bestaan niet enkel uit een groot aantal componenten, de componenten zijn ook meestal verschillend van aard (formalisme). Dergelijke systemen kunnen niet behoorlijk bestudeerd worden door de individuele componenten te bekijken. Het systeem dient in zijn totaliteit bekeken te worden. Het geheel dient als een multi-formalisme systeem behandeld te worden. Hiervoor tracht AToM^3 een oplossing te bieden.

[1] http://moncs.cs.mcgill.ca/MSDL
[2] http://www.cs.mcgill.ca/
[3] http://www.mcgill.ca/

Een manier om het probleem van verschillende formalismen in één systeem op te vangen is door de modelleertaal zelf te modelleren (een meta-model).

Meta-model : Een *meta-model* is een model van de modelleertaal.

Een meta-model beschrijft de mogelijke structuren in een modelleertaal. Het formalisme waarin een meta-model ontwikkeld wordt dient voldoende mogelijkheden te hebben om een modelleertaal te beschrijven. Een vaak gebruikt formalisme voor meta-modellen zijn entity relationship diagrammen [17]. Uitgaande van een meta-model genereert AToM^3 een modelleeromgeving waarin modellen die voldoen aan het meta-model gespecificeerd en gemodelleerd kunnen worden.

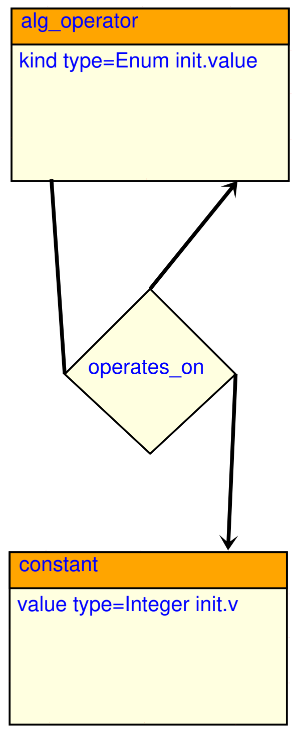
Modellen die voldoen aan het meta-model van het gehanteerde formalisme kunnen dan in AToM^3 worden geëditeerd. De meta-modellen worden in een meta-meta-omgeving gespecificeerd. Dit kan door middel van een entity-relationship diagram [17].
Een model bestaat uit verschillende entiteiten en relaties tussen deze entities. In AToM^3 worden ze beide voorgesteld door objecten.
In figuur 2.1 is een meta-model voorgesteld door een entity relationship diagram in AToM^3. Hierin stellen de rechthoekige objecten entiteiten voor en de ruitvormige objecten relaties tussen de entiteiten. Het entity relationship diagram van figuur 2.1 stelt een model voor waar een algebraïsche operator (`alg_operator`) kan inwerken op een constante (`constant`) of op een algbraïsche operator.
Het diagram beschrijft een boom waar een operator (`alg_operator` in figuur 2.1) kan inwerken op een andere operator of op een constante (`constant` in figuur 2.1).

Een model wordt in AToM^3 intern gerepresenteerd door een ASG (Abstract Syntax Graph) [18] [19] [24].

Abstract Syntax Graaf (ASG): Een abstract syntax graaf is een uitbreiding van de Abstract Syntax Tree (AST).

> **Abstract Syntax Tree:** Een abstract syntax tree is een boom waarin elke knoop een operator is en de kinderen van de knoop de operanden voorstellen [2].

AToM^3 hanteert graafgrammatica's [24] om modellen te transformeren. Een graafgrammatica bestaat uit verschillende regels, elk met een verschillende prioriteit. Een regel bezit een linkerlid (LHS) en een rechterlid (RHS) [19] [18] [20]. Indien in een model het linkerlid van een regel wordt herkend, wordt dit vervangen door het rechterlid van de regel,



Figuur 2.1: Een entity relationship meta-model in AToM^3

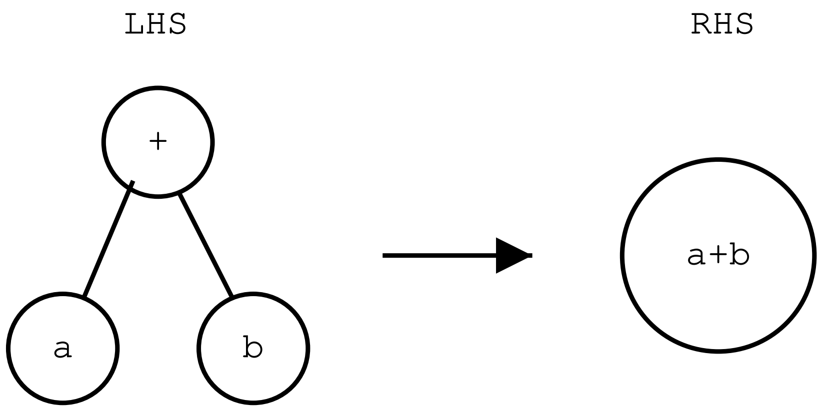
eventueel met uitvoering van geassocieerde acties. Ook moet aan de opgelegde condities voldaan zijn. Als voorbeeld wordt de regel uit figuur 2.2 gegeven. De regel herkent de boomstructuur a+b en vervangt deze door een knoop met als waarde de som van a en b.

2.2 Python

2.2.1 Waarom Python?

Python is vrij makkelijk aan te leren en is erg geschikt voor het snel ontwikkelen van applicaties. De taal is veel eenvoudiger dan C/C++.

- Ten eerste zijn er geen expliciete geheugenallocaties nodig en hiermee samengaand is er een goede ondersteuning voor strings.
- Ten tweede is het een objectgeörienteerde taal wat C niet is.
- Ten derde is AToM^3, de tool waarin de expressies gemanipuleerd zullen worden, geschreven in Python.



Figuur 2.2: Een graafgrammaticaregel

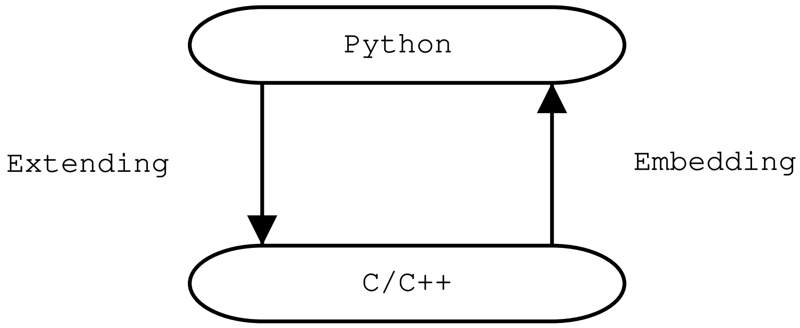
- Ten vierde kan de interpreter interactief gebruikt worden waardoor het mogelijk wordt het ontwikkelde programma op een eenvoudige wijze te testen.
- Ten vijfde is het niet nodig de code telkens opnieuw te compileren.

Wat wel als een voor- of nadeel aanzien kan worden is dat het in Python onnodig is variabelen te declareren. Dit kan zeer handig zijn maar kan tijdens het debuggen van een programma voor verwarring zorgen. Variabelen kunnen tijdens de duur van het programma immers van type veranderen.

Python is een geïnterpreteerde objectgeoriënteerde programmeertaal die vrij beschikbaar is voor de meest courante besturingssystemen (Windows, MacOs, Linux, Unix en andere). Het is een zeer hoge niveau taal met ingebouwde hoge niveau datatypes zoals rijen, dictionaries en lijsten.
De taal maakt gebruik van twee-dimensionale syntax wat tot gevolg heeft dat er geen accolades worden aangewend voor het groeperen van code.
De naam Python verwijst trouwens naar de BBC-serie "Monty Python's Flying Circus" en niet naar een slangensoort.

2.3 Python extensie

Aangezien programmeren in Python eenvoudig is, is het geen slecht idee om een bestaande functionaliteit in een andere programmeertaal op te roepen vanuit Python. Als Python-code gebruik wenst te maken van een C/C++-functie is dit mogelijk. Python heeft de



Figuur 2.3: Extending en embedding van Python

mogelijkheid een *extensie* toe te voegen aan de interpreter. Er wordt een C/C++-module ter beschikking gesteld van Python die kan geïmporteerd worden alsof het een Python-module betrof. De beschikbaar gestelde functies uit de C/C++-extensie kunnen dan in Python gebruikt worden.

De tegenhanger van extending is *embedding* (figuur 2.3). Bij het embedden van Python in C wordt Python-code vanuit een C/C++-programma opgeroepen. Er wordt niet verder ingegaan op deze techniek.

Extending is eenvoudiger, beter gedocumenteerd [9] [7] en kan geautomatiseerd worden met behulp van de tools SWIG [12] en/of CXX [16].
De reden dat embedden moeilijker is dan extenden is vooral te wijten aan het feit dat er geen conventionele manier bestaat om embedding te doen. Dit is vooral het gevolg van de mindere standaardisatie van de programmeertaal C.

2.3.1 Praktisch

Python dient functies uit een C/C++-module op te roepen. Een interface tussen beide talen is bijgevolg noodzakelijk. De interface dient C/C++-data om te zetten naar Python-data en omgekeerd.
Het includeren van "Python.h" in de C/C++-code geeft toegang tot de Python/C API. Deze API converteert Python-data naar C/C++-data en omgekeerd en biedt de mogelijkheid objecten in Python op te roepen [8].
De conversie van data is in handen van de C-programmeertaal.

Een C/C++-file, die dient als code voor een Python-module, moet de volgende onderdelen bevatten:

Oproepbare functies : De *oproepbare functies* zijn de functies die beschikbaar worden gesteld aan Python. Ze worden idealiter `static` gedeclareerd om voor een betere afscherming te zorgen. De functies die deel uitmaken van dit onderdeel geven steeds een pointer naar een python-object (`PyObject *`) terug.

Registratietabel : De *registratietabel* is een C-structuur die de link legt tussen de C/C++-functies en de geassocieerde Python-functies.

Initialisatiefunctie : De *initialisatiefunctie* wordt opgeroepen wanneer Python de module voor de eerste maal importeert. Ze zorgt voor de nodige initialisatie.
Deze functie dient steeds als C geïmplementeerd te zijn. Bij gebruik van C++ is het nodig deze functie `extern ''C''` te definiëren.

De datarepresentatie in de beide talen is uiteraard verschillend. Indien Python een C/C++-functie oproept dienen de doorgegeven argumenten dus omgezet te worden naar C/C++-argumenten. De verantwoordelijkheid voor de omzetting ligt bij de C/C++-implementatie.

Elke oproepbare functie heeft twee argumenten van het type `PyObject *`: `pSelf` en `pArgs`. Het eerste argument wordt enkel gebruikt als de C/C++-functie een ingebouwde methode implementeert. Voor een functie is `pSelf` een `NULL` pointer. Het tweede argument is een pointer naar een Python-tupleobject dat de argumenten bevat van de Python-functie. Elk element van het tuple correspondeert met een argument van de oproepende Python-functie.

Om de argumenten in `pArgs` te kunnen gebruiken in C/C++-code dienen ze omgezet te worden naar C/C++-waarden. Dit kan met behulp van de functie `PyArg_ParseTuple()` uit de Python/C API.

`int PyArg_ParseTuple(PyObject *args, char *format, ...)` : converteert de argumenten van een functie naar lokale variabelen. Deze heeft als return-waarde "waar" bij succes en "vals" als er een fout optreedt. Bij een fout wordt ook een exceptie opgeworpen.

`PyObject *args` : `args` is een tuple object dat een lijst met de argumenten van de Python-functie bevat.

`char *format` : is een formaatstring die aangeeft van welk type de argumenten zijn. Dit kan bijvoorbeeld `''sOi''` zijn, waarvoor het eerste argument van het type `char *`, het tweede van het type `PyObject *` en het laatste van het type `int`.

. . . : hier komen de adressen van de lokale variabelen te staan. Hun type dient overeen te stemmen met de respectievelijke types in de formaatstring.

Om reeds bestaande C/C++-functies te gebruiken in Python kan een *wrapperfunctie* aangewend worden die eerst de argumenten van een Python- naar een C/C++-represtenatie omzet en vervolgens de corresponderende C/C++-functie met C/C++-argumenten oproept. Hiertoe kan de tool SWIG (Simplified Wrapper and Interface Generator) aangewend worden. De wrapperfuncties worden dan voor de gebruiker aangemaakt door SWIG.
Er is niet voor deze manier van werken geopteerd omdat SWIG standaard wrappers produceert die niet steeds voldoen aan de gestelde eisen. De wrapperfuncties werden daarom handmatig gecodeerd.

Voorbeeld

Een voorbeeld verduidelijkt wat hierboven besproken werd. Er wordt nader ingegaan op het bestand `MSLU_atom3_wrapper.cpp`. Dit extensie bestand maakt het mogelijk de MSL-compiler uit hoofdstuk IV op te roepen in Python of AToM^3.

```
// extensie voorbeeld

#include "Python.h"
#include <stdlib.h>
#include <iostream>
#include "MSLUobject.h"
#include "../../back-end/atom3/cpp/PyMSLUobject.h"

extern int yyparse ();
extern FILE* yyin;
extern MSLUOPTR MSLUParsedObject;
using namespace std;

// De oproepbare functie in Python

static PyObject *
MSLUparser (PyObject *pself, PyObject *pargs)
{
  PyObject* inputFile;
```

```
  if (!PyArg_ParseTuple(pargs, "O", &inputFile))
    return NULL;
  yyin = PyFile_AsFile(inputFile);
  if (yyparse() == 1)
    return NULL;
  return (*(PyMSLUObjectClass*)MSLUParsedObject).GetParseObject();
}

//De registratie tabel:

struct PyMethodDef MSLU_parse_methods[] = {
  {"MSLUparser", MSLUparser, METH_VARARGS,
   "parse a MSL USER file"},
  {NULL, NULL}
};

// De initialisatie functie

extern "C"
void
initMSLU_atom3_wrapper ()
{
  (void) Py_InitModule("MSLU_atom3_wrapper", MSLU_parse_methods);
}
```

Deze file is een *wrapper* rond de met behulp van bison gegenereerde parser `yyparse ()`. Bij het importeren van de module in de Python-interpreter wordt de functie `initMSLU_atom3_wrapper` uitgevoerd. Deze roept op zijn beurt `Py_InitModule` op dewelke Python informeert over een nieuwe module met de naam "MSLU_atom3_wrapper".
De bestandsnaam van de module dient overeen te stemmen met de naam van de initialisatiefunctie gebruikt in de code.

De functie `MSLUparser` maakt gebruik van `PyArg_ParseTuple` om de verschillende argumenten uit `PyObject *pArgs` te halen. Het tweede argument van de functie is de formaatstring en de daarop volgende argumenten zijn adressen van lokale variabelen. De formaatstring `''O''` drukt uit dat er één Python-pointer object verwacht wordt.

Compilatie

De gecompileerde C/C++-file wordt niet gebruikt als uitvoerbare code en kan bijgevolg niet op de gebruikelijke wijze gecompileerd worden. Het is nodig het bestand te compileren als een “gedeeld object”. Bij gebruik van de *gcc-compiler* is de optie `-shared` nodig. Het commando dat zorgt voor de compilatie van dit voorbeeld is:

```
gcc -shared -o <target> <source>
```

Onder Windows dient de code gecompileerd te worden als een *.dll bestand.

2.3.2 Callback

Nu het gebruik van een extensie in Python duidelijk is wordt er nog een stap verder gegaan. Vanuit een extensie wordt nu een Python-methode opgeroepen.

In de Python/C API zijn er verschillende functies beschikbaar waarmee Python-functies en -methoden kunnen worden aangesproken vanuit een C/C++-extensie. In dat geval wordt er van een *callback* naar de Python-interpreter gesproken. Voor een functie kan dit met behulp van de functie `PyObject_CallFunction` en voor een methode met `PyObject_CallMethod` [8].

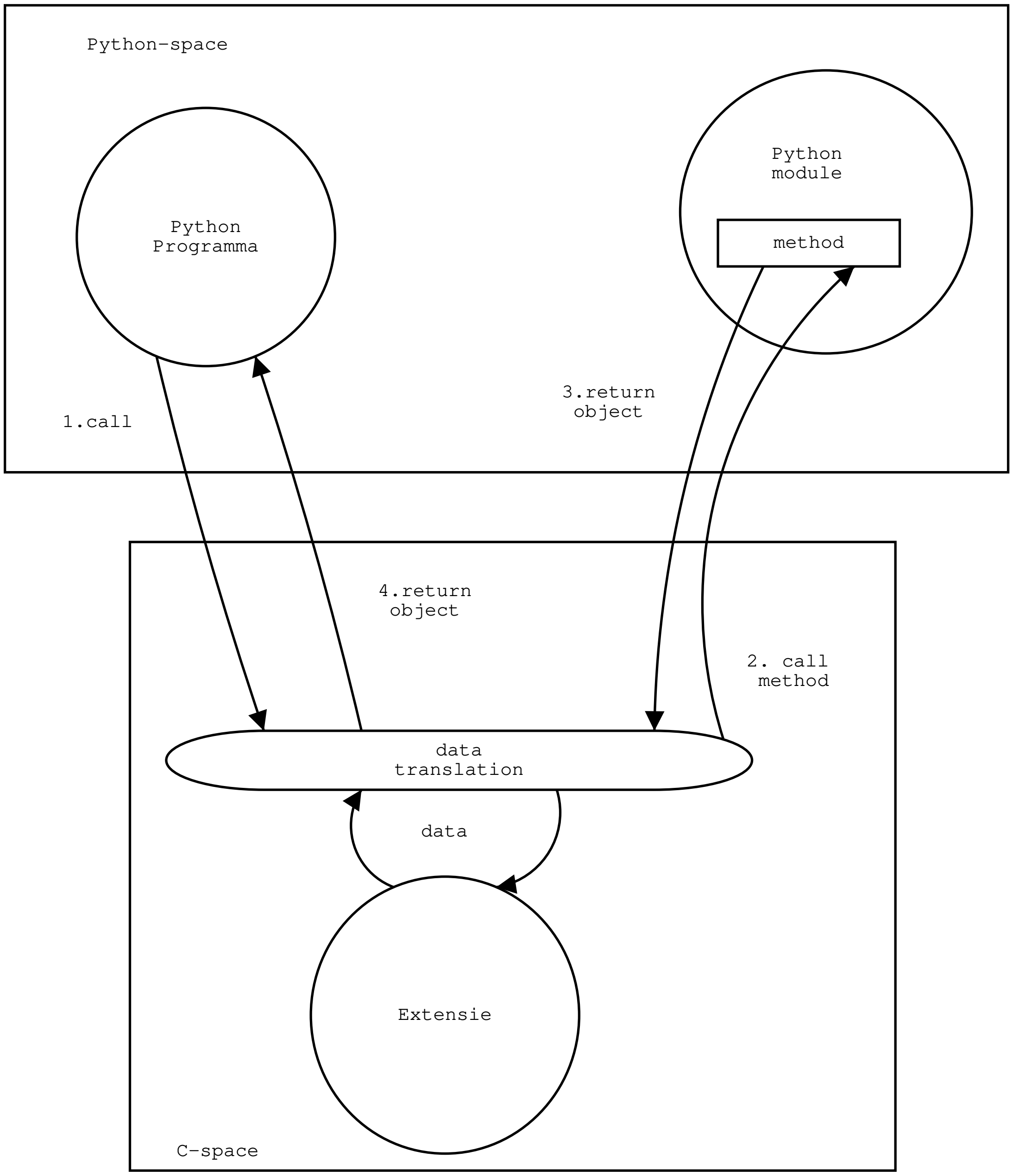
De werkwijze wordt getoond in figuur 2.4. Om het mechanisme duidelijk te maken worden de termen *C-ruimte* en *Python-ruimte* ingevoerd. Met een Python- of C/C++-ruimte wordt het deel bedoeld waarvoor Python respectievelijk C/C++ verantwoordelijk zijn.
In de figuur is een onderdeel *datavertaling* opgenomen. Dit onderdeel zorgt voor de omzetting van Python-data naar C/C++-data en omgekeerd. De datavertaling is de verantwoordelijkheid van de C/C++-ruimte in het mechanisme.

De vertaling geschiedt bij elke overgang van de C/C++-ruimte naar de Python-ruimte en omgekeerd.

Hoe de omzetting van Python naar C/C++ geschiedt kwam reeds in 2.3.1 aan bod.

Het omzetten van C/C++ variabelen naar Python is mogelijk met de functie `Py_BuildValue`.

`PyObject* Py_BuildValue(char *format, ...)` : creërt een nieuwe Python-waarde uitgaande van de formaat string `format` en een reeks waarden.



Figuur 2.4: Callback mechanisme

De verschillende stappen in het communicatieproces worden door figuur 2.4 voorgesteld. De diverse stadia bestaan uit:

1. **oproep**: In de eerste stap wordt vanuit Python een functie (of methode) die in de extensie aanwezig is opgeroepen.
2. **oproep methode**: In de code van de in de eerste stap opgeroepen functie bevindt zich een oproep naar een Python-functie. Deze functie wordt vanuit de C/C++-ruimte in de Python-ruimte opgeroepen.
3. **return object**: De in de vorige stap opgeroepen functie geeft een object terug.
4. **return object**: Het resultaat van de oproep in stap 1 wordt aan het originele Python-programma afgeleverd.

2.4 Besluit

AToM^3 is een modelleer-omgeving die gebruik maakt van meta-modellen om een nieuwe modelleer-omgeving te creëren. De taal gebruikt om AToM^3 te ontwikkelen is Python. Python is een hoge niveau objectgeoriënteerde taal. In Python kunnen extensies gebruikt worden die het mogelijk maken functionaliteit in een andere taal geschreven te gebruiken. Dit mechanisme kan uitgebreid worden naar een callback systeem. In dergelijk callback mechanisme wordt vanuit de niet-Python-code Python-functionaliteit opgeroepen. Dit mechanisme zal worden aangewend in hoofdstuk IV.

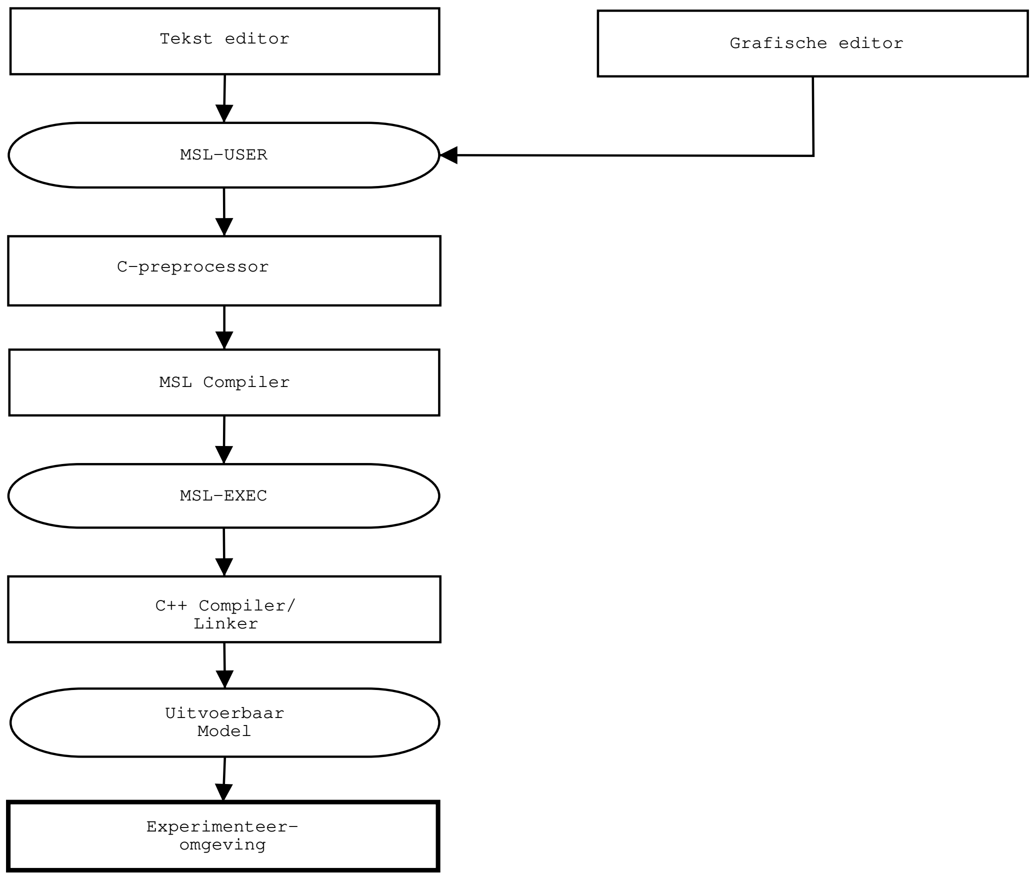
Hoofdstuk III

WEST++

3.1 WEST++

De modelleer- en experimenteer-omgeving WEST++ [33] [32] [26] [28], een acroniem voor "Waste water treatment Environment for Simulation and Training", is een programma dat ontwikkeld werd voor het modelleren van WWTPs (Waste Water Treatment Plants of waterzuiveringsinstallaties). WEST++ wordt nog steeds ontwikkeld aan de vakgroep Toegepaste wiskunde, biometrie en procesregeling van de Universiteit Gent. WEST++ is een niet-commercieel product bedoeld voor onderzoek. De firma Hemmis brengt een commerciële versie, WEST®[29], op de markt. WEST++ wordt op dit moment vooral gebruikt voor het onderzoek (analyse, optimalisatie, ...) van actief slib waterzuiveringsinstallaties maar kan ook voor andere modelleerdoeleinden gebruikt worden. Het geeft de gebruiker een gebruiksvriendelijke omgeving om bestaande modellen (uit een *model base*) te gebruiken of zelf nieuwe modellen te implementeren. Meerdere formalismen worden ondersteund waaronder DAEs (Differential-Algebraic Equations) en Forrester System Dynamics.

In WEST++ is er een strikte scheiding tussen de modelleer- en de experimenteer-omgeving. Daardoor wordt de modulariteit verhoogt en kan de modelleer- of experimenteer-omgeving vervangen worden door een andere tool. Een op zichzelf staande experimenteer-omgeving biedt eveneens het voordeel dat modellen niet vrij beschikbaar moeten gemaakt worden. In een commerciële situatie kan dit belangrijk zijn. De modelleer-omgeving is gericht op de herbruikbaarheid van modellen en de experimenteer-omgeving op het maximaliseren van de nauwkeurigheid en efficiëntie.



Figuur 3.1: Modelleer-proces

Behalve deze twee omgevingen speelt ook de *model base* (modellen bibliotheek) een belangrijke rol in WEST++. In deze model base worden modellen beschreven aan de hand van MSL-USER, een objectgeörienteerde hoge niveau taal. MSL-USER is een variant van MSL (Model Specification Language).

MSL (Model Specification Language) : *MSL* is een krachtige objectgeoriënteerde taal voor de beschrijving van fysische DAE modellen.

3.1.1 Modelleer-omgeving

Het modelleer-proces bestaat uit verschillende onderdelen die opgenomen zijn in figuur 3.1. Hierna volgt een beschrijving van de verschillende stappen.

De WEST++ modelleer-omgeving is een grafische omgeving waar, uitgaande van verschillende componenten, modellen gebouwd kunnen worden (*component-based modelling*). Om dit te bereiken werd een grafische editor ontwikkeld die toelaat interactief modellen op te bouwen. Door het selecteren van verschillende componenten en ze aan elkaar te koppelen wordt een model geconstrueerd. De verschillende beschikbare componenten hebben allen een grafische representatie met één of meerdere inputs en outputs. De in- en outputs worden *terminals* genoemd en worden onderverdeeld in *data terminals* en *fysieke terminals*. Een fysieke terminal stelt een fysieke connectie tussen twee componenten in het model voor. Een connectie die bij het fysiek bouwen van het model ook daadwerkelijk aanwezig is en waarlangs fysieke interactie is. Data terminals staan voor een datastroom tussen componenten. Dit kunnen bijvoorbeeld signalen zijn afkomstig van meetapparatuur die naar een controle-eenheid geleid worden of signalen die van de controle-eenheid komen om een component aan te sturen.

Nadat een model grafisch volledig gebouwd is dient er een model aan elke component verbonden te worden. Een grafische voorstelling zonder enige betekenis zou weinig zinvol zijn. Aan elke component wordt een model uit de model base gekoppeld. Zo een model is gespecificeerd in MSL-USER en bevat het tijdsafhankelijk gedrag van een component.

MSL-USER : *MSL-USER* is de taal die de gebruikers te zien krijgen tijdens het modelleer-proces.

Het geheel van de modellen geassocieerd met de verschillende componenten, de gekozen parameters en de connecties tussen de componenten vormen dan het uiteindelijke model. De selectie van de componentmodellen gebeurt hoofdzakelijk door de WEST++ gebruiker. Er zijn pogingen om automatische modelselectie te doen, gebaseerd op meetresultaten van het echte proces [34].

Na de grafische opbouw van het model en het verbinden van modellen aan de componenten kan de MSL-USER code van het gekoppeld model gegenereerd worden (zie figuur 3.1). MSL-USER code kan ook onmiddellijk door de gebruiker in een tekst editor worden ingevoerd.

De MSL-USER code van een model is nu beschikbaar en wordt door een C-prepocessor behandeld. De C-preprocessor voert tekstuele manipulaties uit op de MSL-USER code. In een MSL-USER bestand (meestal met extensie .msl) kan behalve MSL-USER-code ook C++ code opgenomen worden. Op deze manier kunnen sommige dingen (bijvoorbeeld een sinus) eenvoudiger voorgesteld worden.

De gecreëerde MSL-USER code wordt door de MSL compiler omgezet naar MSL-EXEC.

MSL-EXEC : *MSL-EXEC* is de op C++ gebaseerde uitvoerbare variant van MSL.

Een uitvoerbaar MSL-EXEC model bestaat uit:

- Hiërarchische symbolische informatie omtrent in het model gebruikte variabelen en parameters.
- Een aantal standaardfuncties die de vergelijkingen uit het model bevatten:
  - `ComputeInitial()` : Deze functie bevat vergelijkingen die slechts eenmaal, bij het opstarten van de simulatie, moeten doorgerekend worden.
  - `ComputeState()` : Deze functie is de meest omvangrijke en omvat de algebraïsche- en differentiaalvergelijkingen die de toestand van het model bepalen.
  - `ComputeTerminal()` : Indien er vergelijkingen zijn die enkel bij het beëindigen van de simulatie moeten worden doorgerekend, komen die in deze functie terecht.

De door de MSL compiler gegenereerde code (MSL-EXEC) wordt door een C++ compiler/linker gecompileerd en gelinkt tot binaire code. De bekomen binaire code kan door de experimenteer-omgeving worden ingeladen.

De zelfgebouwde modellen zijn herbruikbaar. De gekoppelde modellen beschikken, zoals de componenten waaruit ze zijn opgebouwd, over een interface. Een geconstrueerd model kan worden toegevoegd aan de model base en herbruikt worden in nieuwe gekoppelde modellen. Daardoor ontstaat een hoge graad van herbruikbaarheid. Vanuit een praktisch standpunt is dit dan ook voordelig. Delen van een model kunnen eerst afzonderlijk ontwikkeld en getest worden. Als ze voldoende stabiel zijn kunnen ze aan elkaar gekoppeld worden met behulp van de grafische editor.

3.1.2 Experimenteer-omgeving

Eens een model volledig gespecificeerd is in de grafische editor wordt het in de modelleeromgeving omgezet naar binaire code (zie 3.1). Deze binaire code wordt gebruikt in de experimenteer-omgeving van WEST++.

Met behulp van de experimenteer-omgeving kan de modelbouwer virtuele experimenten uitvoeren op de gecompileerde versie van zijn model. De experimenteer-omgeving kan onder andere een numerieke simulatie berekenen die als output het toestandstrajectorie van het model geeft, uitgaande van het MSL-EXEC model. Daarbij kan gebruik gemaakt worden van verschillende numerieke oplossingstechnieken.

In WEST++ kunnen verschillende virtuele experimenten worden uitgevoerd:

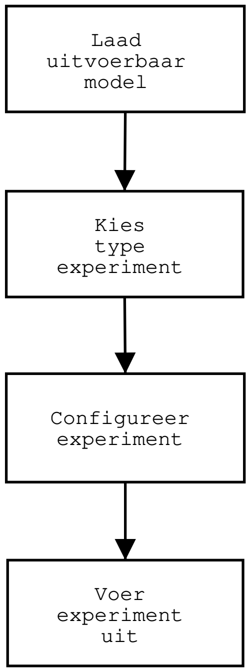
1. Simulatie
2. Trajectorie optimalisatie
3. Eindwaarde optimalisatie
4. Sensitiviteits analyse
5. Monte Carlo simulatie

Voor meer informatie in verband met deze mogelijke virtuele experimenten wordt verwezen naar [26].

Het verloop van het experimenteer-proces is grafisch voorgesteld in figuur 3.2. De verscillende stapppen zijn:

- **Laad uitvoerbaar model** : De binaire code afkomstig van de modelleer-omgeving (3.1.1) wordt ingeladen.
- **Kies type Experiment** : Één van de mogelijke virtuele experiment types wordt gekozen.
- **Configureer experiment** : De relevante configuraties (instellen parameter waarden, instellen initiële condities, ...) van het experiment worden uitgevoerd.
- **Voer experiment uit** : Het virtueel experiment wordt op basis van de gekozen instellingen uitgevoerd.

Een andere aantrekkelijke eigenschap van de WEST++ simulatie-omgeving is het gebruik van een scripting taal (Tcl scripting en Visual Basic scripting) om simulaties te automatiseren.



Figuur 3.2: Experimenteer-proces

3.1.3 MSL-USER 4.0

MSL-USER 4.0 is een objectgeoriënteerde taal die toelaat om de dynamica van een model op een declaratieve manier te beschrijven. Declaratief wil zeggen dat wordt gecodeerd hoe een model er uitziet en niet hoe het dient opgelost te worden.

In MSL-USER komen de meeste objectgeorienteerde begrippen terug. Er wordt gebruik gemaakt van types en objecten om de hiërarchie van de verschillende objecten in de model base te beschrijven.

Hierna volgt een beschrijving van de MSL-USER 4.0 taal die door de MSL compiler uit hoofdstuk IV herkend wordt. De volledige syntax is te vinden in appendix A.

Een MSL-USER 4.0 bestand bestaat uit een object, het *topobject*. In dit topobject kunnen andere objecten gedefinieerd worden.

```
<MSL file> ::= <topObject>
             |
<topObject> ::= <object>
<object> ::= <explicitObject>
           | <implicitObject>
```

```
<explicitObject> ::= <objectdeclaration> := <object>
                   | <objectdeclaration>
<objectdeclaration> ::= OBJ <identifier> <optionalComment> : <type>
<implicitObject> ::= <typedeclaration>
                   | <equationObject>
                   | <expressionObject>
```

Onder objecten vallen, naast een gewone objectdeclaratie (`<objectdeclaration>`) ook typedeclaraties (`<typedeclaration>`), objecten die vergelijkingen tussen objecten voorstellen (`<equationObject>`) en objecten die expressies representeren (`<expressionObject>`). Declaraties van het type `CLASS` (MSL-USER 3.1) zijn in MSL-USER 4.0 niet mogelijk. Hiervoor wordt `OBJ` gebruikt.
Een `<object>` kan verschillende vormen aannemen bijvoorbeeld ¡setObject¿ of ¡productObject¿. Voor alle mogelijke vormen van een `<object>` wordt de lezer verwezen naar appendix A.

Typedeclaraties

Een typedeclaratie heeft de vorm:

```
<typedeclaration> ::= TYPE <type>
 | TYPE <identifier> <optionalComment> = <type> <optionalDefaultObject>
```

Door gebruik te maken van `TYPE <identifier> <optionalComment> = <type> <optionalDefaultObject>` kunnen, uitgaande van bestaande types, nieuwe types gedeclareerd worden.
In MSL-USER zijn een aantal types standaard aanwezig:

```
<type> ::= <basicType>
         | <setType>
         | <productType>
         | <vectorType>
         | <operatorType>
         | <unionType>
         | <referenceType>
```

```
          | <bracketedType>
          | <identifierType>
```

Deze *basistypes* zijn beschikbaar in MSL-USER 4.0:

```
<basicType> ::= Bottom
              | Generic
              | TypeType
              | Integer
              | Real
              | Character
              | String
              | <enumeratedType>
```

Hun syntax en semantiek zijn:

Bottom type :

- Syntax: `TYPE Bottom`
- Semantiek: Het Bottom type is een subtype van elk type. `NULL` is van het type Bottom.

Generic type :

- Syntax: `TYPE Generic`
- Semantiek: Een variabele van het type Generic is unificeerbaar met elk ander type. Elk type is een subtype van het type Generic.

Integer type :

- Syntax: `TYPE Integer`
- Semantiek: $\in \mathbb{Z}$

Real type :

- Syntax: `TYPE Real`
- Semantiek: $\in \mathbb{R}$

Character type :

- Syntax: TYPE Character
- Semantiek: Is een karakter.

String type :

- Syntax: TYPE String
- Semantiek: Is een string.

enumerated type :

- Syntax: <enumeratedType> ::= ENUM <id$_1$>, ..., <id$_n$> END_ENUM
- Semantiek: Een identifier van het type enumerated is equivalent aan een constante integer waarde met als waarde de index van de identifier in de declaratielijst.

De volgende *samengestelde types* zijn voor handen in MSL-USER 4.0:

set type :

- Syntax: SET <type> END_SET
- Semantiek: Bepaalt een verzameling met elementen van het type <type>.

product type :

- Syntax: PRODUCT <type$_1$>, ..., <type$_n$> END_PRODUCT
- Semantiek: Bepaalt het type <type$_1$> x ... x <type$_n$>.

vector type :

- Syntax: <orientation> VECTOR <optionalRange> OF <type> END_VECTOR
  <orientation> ::= ROW | COLUMN |
- Semantiek: Een rij- of kolomvector, afhankelijk van <orientation> (default ROW), van grootte <optionalRange> en type <type>.

operator type :

- Syntax: <OPERATOR <optionalType$_1$> -> <optionalType$_2$> END_OPERATOR

- Semantiek: Het type van een operator met als type van het domein <optionalType$_1$>. en als type van het beeld <optionalType$_2$>.

union type :

- Syntax: UNION <type$_1$>, ..., <type$_n$> END_UNION
- Semantiek: Objecten van het type UNION kunnen elk type in <type$_1$>, ..., <type$_n$> aannemen.

reference type :

- Syntax: REF <type> END_REF
- Semantiek: Objecten van dit type zijn een referentie naar een object.

bracketed type :

- Syntax: (<type>)
- Semantiek: Gebruikt om de precedentie expliciet te bepalen.

identifier type :

- Syntax: <identifier>
- Semantiek: De naam van een door de gebruiker gedefinieerd type.

De basistypes van MSL-USER 4.0 stemmen overeen met de basistypes in MSL-USER 3.1. Enkel het type Boolean is niet beschikbaar in MSL-USER 4.0. Het Boolean type kan vervangen worden door variabelen verschillend van 0 als "waar" te interpreteren en variabelen gelijk aan nul als "vals".
De samengestelde types record type en powerset type uit MSL-USER 3.1 zijn niet meer beschikbaar in MSL-USER 4.0. Het type function type veranderde van naam en is in MSL-USER 4.0 het operator type. De samengestelde die types die bruikbaar zijn in MSL-USER 4.0 veranderden wel van syntax ten opzichte van de deze in MSL-USER 3.1 [33].

3.1.4 Besluit

WEST++ is een modelleer- en simulatie-omgeving die vooral gebruikt wordt voor het modelleren en simuleren van waterzuivering. Het programma bestaat uit twee hoofdcomponenten: een modelleer-omgeving en een experimenteer-omgeving. Deze zijn van elkaar gescheiden om de modulariteit te verhogen.

De modellen die gekoppeld worden aan de componenten van een model worden in MSL-USER gecodeerd en samen gebracht in een model base.

MSL-USER 4.0 is een objectgeoriënteerde taal die toelaat om de dynamica van een model op een declaratieve manier te beschrijven.

Eens een model volledig gespecificeerd is in de modelleer-omgeving wordt het door de parser omgezet naar MSL-EXEC. De MSL-EXEC code kan vervolgens gecompileerd worden en verschillende soorten virtuele experimenten kunnen dan uitgevoerd worden binnen de WEST++ omgeving.

Hoofdstuk IV

MSL parser

4.1 Inleiding

In hoofdstuk I werden de verschillende fasen van een compilatieproces besproken. Het programma WEST++ en zijn verschillende onderdelen werden in hoofdstuk III van naderbij bekeken. In hoofdstuk II werd dieper ingegaan op de meta-modelleeromgeving AToM^3 en het gebruik van Python en Python-extensies. In dit hoofdstuk zullen deze drie onderdelen samengebracht worden voor de ontwikkeling van een nieuwe MSL compiler.

Bij de ontwikkeling en implementatie van software worden er vaak fouten gemaakt en zijn er bijna altijd bugs. Om die reden is gekozen de compiler stapsgewijze (prototypes) op te bouwen. Een eerste prototype (4.2) wordt gebruikt om de tools flex en bison te leren kennen. Het tweede prototype (4.3) maakt gebruik van een Python-extensie. Het tweede prototype wordt eveneens geïntegreerd in AToM^3. Daarna wordt de MSL compiler geïmplementeerd gebruik makende van flex, bison en Python.

In sectie 4.2 wordt een eerste prototype compiler geïmplementeerd op basis van een eenvoudig flex- en bison-specificatie. In sectie 4.3 wordt de parser uitgebreid om een object in Python op te bouwen waarbij gebruik gemaakt wordt van een extensie (2.3) en het callback mechanisme (zie 2.3.2). Sectie 4.4 wendt de techniek uit 4.3 aan om de nieuwe MSL compiler te bouwen.

4.2 Prototype 1

De tools flex en bison worden in deze sectie voor de eerste keer gebruikt. Om die reden wordt eerst een eenvoudige compiler geïmplementeerd.

In het eerste prototype werd een compiler gebouwd die algebraïsche expressies bestaande uit optellingen, vermenigvuldigingen en negaties kan vertalen naar Python-code. Die code kan geïnterpreteerd worden door AToM^3.

De algebraïsche expressies vormen op zichzelf een model. In AToM^3 wordt het meta-model algebra_entities.py gebruikt (zie figuur 2.1) om te specificeren wat de vorm van de expressies dient te zijn. Een voorbeeld van een mogelijke expressie:

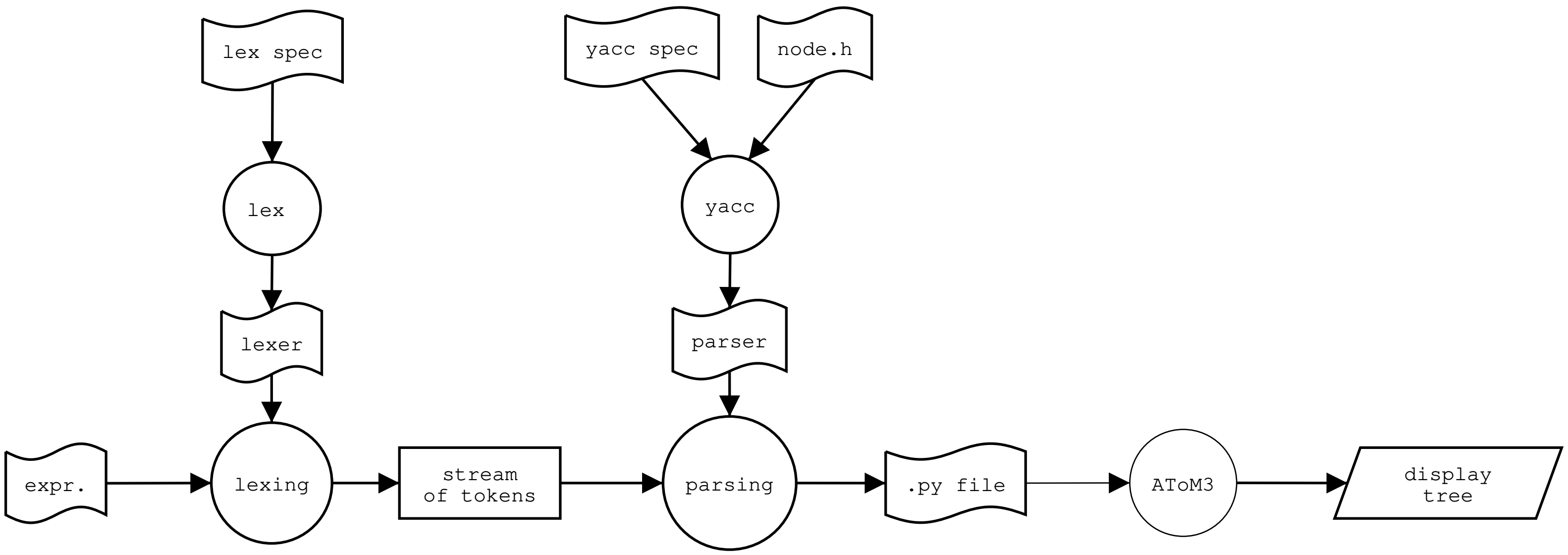
$$(7*(-8)*((-6)*(5*(99*6))))*(((5*33)*3)*9*45*78*96*9*5)$$

De verschillende fasen van het compilatieproces zijn opgenomen in figuur 4.1. Er wordt een lexicale en grammaticale specificatie opgesteld. Uit deze specificaties worden de lexer en de parser geproduceerd door gebruik te maken van de tools flex en bison. Op het invoerbestand (hierin staan de expressies) wordt eerst een lexicale analyse toegepast. De stroom van tokens, die de lexicale analyse genereert, wordt verwerkt door de grammaticale analyse. Tijdens de grammaticale analyse wordt een abstract syntax tree opgebouwd. Daarna wordt er Python-code, de vertaling van het invoerbestand, naar een bestand geschreven. Hiervoor wordt de abstract syntax tree gebruikt. Deze Python-code is interpreteerbaar door AToM^3.

Een model van een algebraïsche expressie wordt door de compiler omgezet naar Python-code. De file die de Python-code bevat kan door het programma AToM^3 geïnterpreteerd worden. Door de gegenereerde file te openen in AToM^3 komt een grafische representatie van de gecompileerde algebraïsche expressie tot stand. De Python-code zorgt voor de creatie van de objecten en het leggen van de connecties tussen de objecten in AToM^3, wat wordt weergegeven in figuur 4.4.

In AToM^3 worden modellen intern voorgesteld door een ASG (zie 2.1). De kinderen van een knoop in een graaf hebben geen volgorde. In AToM^3 is het de volgorde van creatie van de objecten die impliciet een volgorde aan de objecten van de ASG geeft.

De gebruikte bewerkingen (vermenigvuldiging en optelling) in het meta-model (figuur 2.1) zijn commutatief wat met zich meebrengt dat de volgorde van de kinderen van geen belang is. Maar in 4.2.5 blijkt het toch voor een mogelijk probleem te zorgen. Door het herhaaldelijk aanbieden van dezelfde uitdrukking zou een lexicaal verschillende uitdrukking



Figuur 4.1: Compilatie van een model

bekomen kunnen worden omdat de volgorde van de knopen in de ASG anders zou kunnen zijn.

Een model van een algebraïsche expressie wordt door de compiler omgezet naar Python-code. De Python-code zorgt voor de creatie van de objecten en het leggen van de connecties tussen de objecten in AToM^3.

De tools flex en bison werden aangewend om de compiler, die een model omzet naar Python-code voor gebruik in AToM^3, te coderen.

4.2.1 Lexicale analyse

De gebruikte lexicale specificatie werd reeds als voorbeeld aangehaald in 1.4.1. Er wordt dus ook naar die sectie verwezen voor de code.

4.2.2 Grammaticale analyse

De aangewende grammatica is een traditionele grammatica die geen gebruik maakt van expliciete declaratie van precedenties of associativiteit. Deze kunnen afgeleid worden uit de grammatica, wat de duidelijkheid van de grammatica ten goede komt. Het gebruik van expliciete declaraties zou hier de verstaanbaarheid slechts op geringe wijze beïnvloeden. Indien aan de grammatica producties worden toegevoegd kan de duidelijkheid snel verdwijnen bij expliciete precedentiebepalingen.

De gebruikte contextvrije grammatica, opgenomen in figuur 4.2, bestaat uit:

1. Een stel tokens of terminale symbolen (afkomstig van de lexer).
2. Een stel niet-terminalen.
3. Een stel producties die elk bestaan uit een niet-terminale linkerzijde, een pijl en een een sequentie van tokens en/of niet-terminalen, de rechterzijde.
4. Een niet-terminaal symbool dat als startsymbool wordt aangeduid.

Productie: Een productie is van de vorm `A` $\rightarrow$ `B` waar de pijl de betekenis heeft "A kan de vorm B hebben".

$$START \rightarrow expression \quad (4.1)$$
$$expression \rightarrow expression\, PLUS\, term \quad (4.2)$$
$$\mid term \quad (4.3)$$
$$term \rightarrow term\, MUL\, factor \quad (4.4)$$
$$\mid factor \quad (4.5)$$
$$factor \rightarrow '(' expression ')' \quad (4.6)$$
$$\mid INV\, number \quad (4.7)$$
$$\mid number \quad (4.8)$$
$$number \rightarrow NUMBER \quad (4.9)$$

Figuur 4.2: Producties prototype 1

Gemakkelijkheidshalve worden producties met dezelfde linkerzijde meestal samen genomen. Tussen de verschillende rechterzijden wordt dan een "|" geplaatst met als betekenis "of".

Tijdens het parsen wordt een binaire abstract syntax tree opgebouwd die de expressie in het model voorstelt. Deze binaire abstract syntax tree bestaat uit knopen en bladeren die van het type "`node`" zijn. De datastructuur "node" wordt beschreven in 4.2.3.

De volledige yacc-code is samen met de code voor de `main` functie te vinden in appendix B.

De main functie genereert onder andere de Python-code die de geparste expressie in AToM^3 voorstelt.

4.2.3 Datastructuur

De datastructuur die aangewend wordt voor de voorstelling van de knopen en bladeren van de boom slaat het type, het nummer en de coördinaten op. Afhankelijk van het type worden de operanden van de knoop of de waarde van een blad bewaard. Het type van een knoop kan T_MUL (indiende operator * is), T_PLUS (indien de operator + is) of T_INV (negatie) zijn. Het type van een blad is T_NUMBER wat er op wijst dat het blad een positief geheel getal bevat. De operanden van een knoop zijn pointers naar dezelfde datastructuur. Hierdoor wordt het eenvoudig de binaire boom bottom-up te creëren. De boom wordt na het parsen van de expressie diepte-eerst doorlopen. Bij elke knoop en blad wordt de nodige Python-code naar een bestand geschreven.

```
struct node
{
  enum type node_type;              // type van de knoop
  unsigned node_nr;                 // nummer van de knoop
  int X_COOR, Y_COOR;               // coördinaten van de knoop
  union data_union
  {
    struct op_node
    {
      struct node *leftarg_ptr;     // linker operand
      struct node *rightarg_ptr;    // rechter operand
    }
    op;
    int value;                      // waarde van een T_NUMBER knoop
  }
  data;
};
```

Figuur 4.3: C-datastructuur

De gebruikte datastructuur in C-code is opgenomen figuur 4.3.

4.2.4 Plaatsing objecten

Het toekennen van de coördinaten aan de verschillende objecten van een model gebeurt zodanig dat er geen overlapping bestaat tussen de verschillende objecten.

Voor het compileren wordt gebruik gemaakt van yacc. Dit is een LR(1) parser en werkt bijgevolg bottum-up. De bladeren van de binaire boom worden dan het eerst gecreëerd. Voor een expressie in het model zijn deze bladeren de gehele getallen van het model. Uitgaande van deze eigenschap werd een eenvoudig algoritme ontwikkeld dat voor een horizontale of verticale lay-out van de boom zorgt.
De manier waarop het toekennen van de coördinaten aan de verschillende knopen en bladeren plaatsvindt wordt hierna beschreven voor de verticale representatie van de boom. Het geval van de horizontale representatie is analoog.

Tijdens het parsen worden aan de bladeren stijgende x-coördinaten gegeven die ervoor zorgen dat de bladeren niet kunnen overlappen. Een binaire operatorknoop krijgt als x-coördinaat de helft van de som van de x-coördinaten van zijn kinderen. Een unaire operatorknoop krijgt als x-coördinaat dezelfde x-coördinaat als zijn kind. Na het parsen van de expressie wordt de opgebouwde parse-boom top-down doorlopen en worden de y-coördinaten stijgend per niveau toegekend. Dit zorgt voor een grafische voorstelling die geen overlappingen geeft en er visueel aanvaardbaar uitziet. De verschillende objecten staan niet nodeloos ver uit elkaar zodat er niet overdreven veel plaats wordt ingenomen.

In pseudo-code:

```
Y_START = 35
Y-coordinate_root = Y_START
node_size = 70
i = 1
VSPACE = 115

if node_type == T_NUMBER
  X-coordinate = i * node_size
  i++
elif node_type == T_INV
  X_coordinate = X_coordinate_child
elif node_type == T_MUL or node_type == T_PLUS
  X-coordinate =
          (X-coordinate_left_arg + X-coordinate_right_arg) / 2

traverse tree top-down
  Y-coordinate = Y-coordinate_previous_level + VSPACE
```

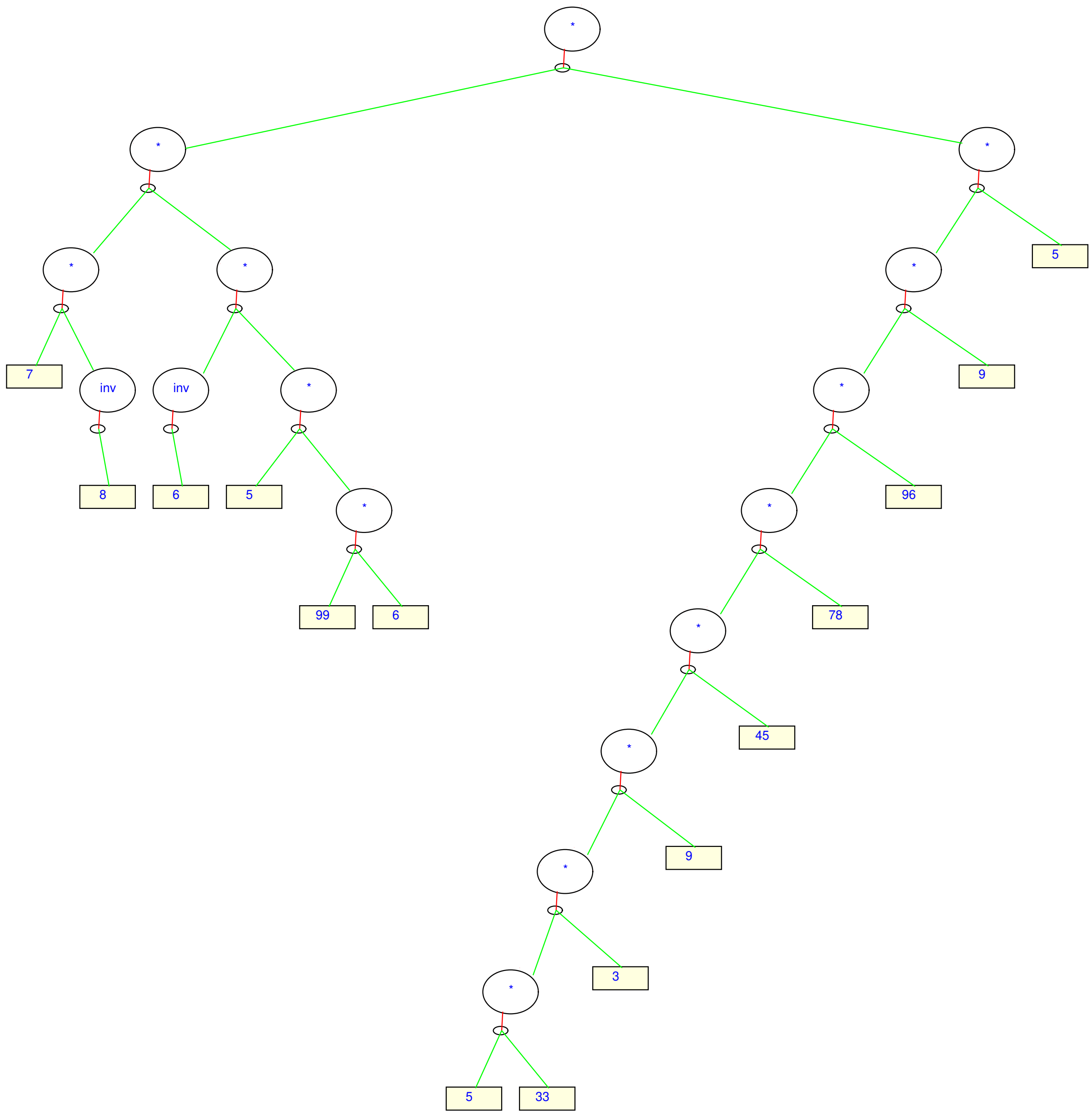
Een voorbeeld is de uitdrukking:

$$(7 * (-8) * ((-6) * (5 * (99 * 6)))) * (((5 * 33) * 3) * 9 * 45 * 78 * 96 * 9 * 5)$$

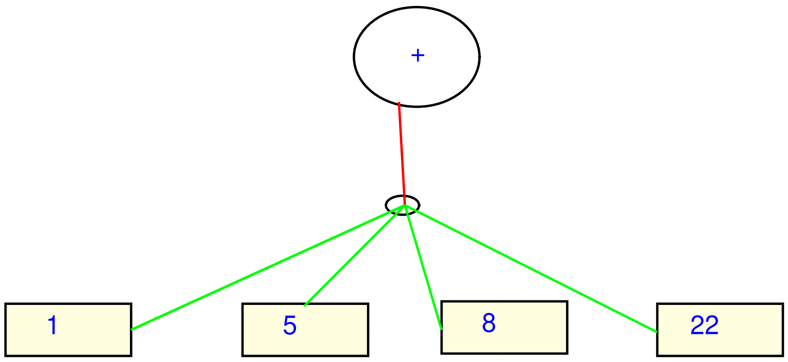
, waarvan de plaatsing van de verschillende objecten te zien is in figuur 4.4.

4.2.5 Expressies in AToM^3

AToM^3 hanteert een graafgrammatica om modellen te transformeren. Voor het uitschrijven van een expressie in zijn originele vorm naar een bestand wordt echter geen gebruik



Figuur 4.4: Plaatsing objecten



Figuur 4.5: N-aire operator

gemaakt van een graafgrammatica. Het uitschrijven van de expressie met behulp van een graafgrammatica zou de boom volledig kopiëren. Bij uitvoering van een graafgrammaticaregel zou telkens een deel van de uitdrukking gegenereerd worden. Graafgrammatica's zijn echter niet efficiënt. Het zoeken naar een subgraaf in een graaf is namelijk een NP-compleet probleem. Om die reden wordt het uitschrijven van de expressie dan ook hard gecodeerd.

In de gehanteerde oplossing wordt de boom diepte-eerst behandeld. In AToM^3 hebben de kinderen van een knoop geen volgorde. Het doorlopen van de boom kan bijgevolg lexicaal verschillende expressies opleveren. Dit probleem lost zichzelf echter op omdat AToM^3 impliciet een volgorde aan de objecten geeft.

In een boom die correspondeert met een expressie worden geen haakjes waargenomen, zelfs indien ze van belang zijn in de uitdrukking. De sequentie van de bewerking is afleidbaar uit de boom. Bij het uitschrijven mag dit niet uit het oog verloren worden.
Aan de verschillende operatoren worden precedenties toegekend. Uit de precedenties van elkaar opvolgende operatoren wordt afgeleid of haakjes al dan niet noodzakelijk zijn. Op deze manier wordt het aantal haakjes tot een minimum herleid.

Het algoritme voor het uitschrijven van expressies is tot meer in staat dan de compiler.
Meerdere bomen in een model zijn mogelijk. Voor elke boom wordt een expressie naar een bestand geschreven. De + en *-operator zijn in het algebra_entities.py model geen binaire maar n-aire operatoren. N-aire operatoren vormen tevens geen probleem voor het algoritme (figuur 4.5 stelt de uitdrukking 1+5+8+22 voor). Ook de negatie van een uitdrukking kan behandeld worden door de code.

```
# class that implements the nodes that form the tree

class node(object):
  '''representation of a node'''

  def __init__(self):
     self.node_nr = -1             # the number of nodes that have been created
     self.x_coor = -1              # the x coordinate
     self.y_coor = -1              # the y coordinate
     self.direction = None         # the direction of the tree
```

Figuur 4.6: De node klasse

4.3 Prototype 2

Het tweede prototype bouwt verder op het eerste waarbij de lexicale en grammaticale specificaties behouden blijven.
Het opzet van dit prototype is het construeren van een object in Python in plaats van een structuur in C/C++. Dit Python-object kan onmiddellijk in AToM^3 gebruikt worden. De extra stap, het schrijven naar een bestand, van het vorige prototype (sectie 4.2) is niet meer nodig. Hiervoor wordt de Python interpreter voorzien van een *extensie* (zie sectie 2.3) die de compiler vanuit AToM^3 oproept.

4.3.1 De `node` klasse

De boom die een expressie voorstelt wordt niet meer als een C-`struct` opgebouwd maar wel als een object van de Python-klasse "`node`" (figuur 4.6). Een object van deze klasse beschikt over de attributen `node_nr`, `x_coor` en `y_coor`.

Van deze klasse worden twee nieuwe Python-klassen afgeleid:

- `number_node` : Objecten van de klasse `number_node` (figuur 4.7) vormen de bladeren van de boom. Deze objecten stellen de constanten van de expressie voor en krijgen een extra attribuut `value`.

- `operator_node` : Objecten van de klasse `operator_node` (figuur 4.8) zijn de knopen van de boom. Instanties van deze klasse stemmen overeen met operatoren uit de

```
class number_node(node) :
  '''class representing a number node'''

  def __init__(self):
     self.value = None
```

Figuur 4.7: De number_node klasse

```
class operator_node (node) :
  '''class representing an operator node'''

  def __init__(self):
     self.operator = None
     self.operands = [ ]
```

Figuur 4.8: De operator_node klasse

expressie. De operanden (objecten van de klasse `number_node` of `operator_node`) worden in het attribuut `operands` als items van een lijst opgeslaan.

Beide kinderklassen beschikken over een aantal extra methoden ten opzichte van hun ouderklasse.
Voor de twee klassen worden de methoden `write` en `atom3_build` geïmplementeerd.
Voor de klasse `operator_node`-klasse is er naast deze twee methoden nog een methode `add_operand` voorzien.
De functionaliteit van de verschillende methoden is:

- `write`: De methode `write` schrijft het `node`-object en zijn eventuele operanden uit als een traditionele expressie naar de standaard output.

- `atom3_build`: De methode `atom3_build` zorgt ervoor dat het `node`-object wordt vertaald naar een AToM^3-object.
  De methode heeft enkel zin als ze in AToM^3 met de juiste argumenten wordt opgeroepen.

- `add_operand`: De methode `add_operand`, die enkel voorkomt in de klasse `operator_node`, voegt een operand toe aan het data-member, een instantie van de klasse `operator_node`.

4.3.2 Integratie met AToM^3

De compiler bouwt nu een Python-object van het type `node` op. Het initiële doel is echter het oproepen van de compiler in AToM^3.
Het gebruik van een knop in de AToM^3 die de compiler oproept is hier de aangewezen manier.

Bij het indrukken van de knop is er de keuze tussen het parsen van een file of een string. Dan is er de keuze de boom verticaal of horizontaal op het scherm te tekenen.
Na het lezen van een expressie wordt een object van het type `node` gecreëerd door gebruik te maken van de methode `expr_parse` (aanwezig in de module *msl.so*). Deze methode start het parsen van de expressie.
Eens het parsen beëindigd is, is het bekomen object niet bruikbaar in AToM^3. Door gebruik te maken van de methode `atom3_build`, uit de klasse `node`, kan uit het object de grafische en semantische betekenis voor het AToM^3 meta-model bekomen worden.

4.4 MSLU 4.0 parser

In deze sectie zal een nieuwe MSL compiler ontwikkeld worden. De op dit moment beschikbare MSL compiler is te beperkend voor de nieuwe modellen die moeten gesimuleerd worden. Bij het parsen van matrices worden bijvoorbeeld alle elementen van de matrix omgezet naar een `#define` statement. Bij grote matrices stijgt het aantal `#define`s en de corresponderende `#undef`s zeer snel. Als het aantal te groot wordt ontstaan er problemen bij de compilatie van de MSL-EXEC file.

4.4.1 Distributie

De distributie die alle programmabestanden bevat is te vinden op: http://msdl.cs.mcgill.ca/MSDL/people/tmcornel/thesis/MSL/MSL_html.

Tijdens de implementatie van alle code is veel aandacht besteed aan het platform onafhankelijk zijn van de geïmplementeerde code. De compilatie op Linux en Windows is mogelijk. Bij gebruik van het Linux besturingssysteem is gebruik gemaakt van de gcc compiler versie 3.2. Onder Windows diende Microsoft Visual C++ 6.0 als compiler.

Om het geheel overzichtelijk te maken is alles opgenomen in een directory-structuur (zie figuur 4.9). De drie belangrijkste directories zijn `bin`, `build` en `src`.

Alle code is te vinden in de `src` directory. De verschillende bestanden zijn per logisch onderdeel in verschillende directories ondergebracht. De bestanden die tot de front-end behoren komen in `src/front-end` en die van de back-end zijn te vinden in `src/back-end`. In `src/wrapper` zijn twee directories, `src/wrapper/atom3` en `src/wrapper/westpp`, te vinden waarin code staat die een wrapper rond de compiler legt. De front-end en de back-end worden namelijk als een bibliotheek gecompileerd en kunnen op die manier op verschillende plaatsen gebruikt worden. De bestanden in `src/wrapper/westpp` zijn de implementatie van een wrapper die een uitvoerbaar programma van de compiler produceren. In de directory `src/wrapper/atom3` staat de implementatie die de compiler als Python-extensie compileert. De overige twee directories in de `src` directory, `atom3` en `MSLE`, bevatten Python-bestanden die bestemd zijn voor het gebruik van de compiler in AToM^3 en voor het uitschrijven van MSL-EXEC code.

De code in de `src` directory dient natuurlijk gecompileerd te worden. De bestanden die daarvoor zorgen zijn te vinden in de `build` directory. Deze directory wordt onderverdeeld in `build/linux` en `build/win32-msvc6.0`. De eerste directory zorgt voor de compilatie onder Linux en de tweede voor de compilatie onder Windows. Binnen deze twee directories bestaat er een één-één relatie met de directories in `src` die C/C++ code bevatten.

In een Linux omgeving wordt er gebruik gemaakt van `make` [13] om alles te compileren. Op elk niveau in de directory-structuur is er een `makefile` aanwezig die zorgt voor de compilatie van de overeenstemmende bestanden in de `src` directory.
Bij gebruik van Microsoft Visual C++ zijn het bestanden met extensie `.dsw` en `.dsp` die zorgen voor de configuratie van de programmeeromgeving. Er dient dan nog enkel een build van het project gedaan te worden.

In de `doc` directory is documentatie over de C/C++ code opgenomen. Om de documentatie te genereren werd de tool *Doxygen* [31] gebruikt. Doxygen produceert uitgebreide documentatie van de gebruikte files en de inhoud van de files.

De `etc` directory bevat het configuratiebestand voor Doxygen en voor het Linux platform. In het bestand `etc/linux/MSLU.conf.sh` zijn verschillende systeemvariabelen gedefinieerd.

De gecompileerde bestanden komen afhankelijk van het gebruikte platform terecht in de directories `lib/linux`, `lib/win-msvc6.0`, `bin/linux` of `bin/win-msvc6.0`.

```
$MSLU_ROOT_PATH
  /bin
    /linux
    /win32-msvc6.0
  /build
    /linux
      /back-end
        /atom3
          /cpp
        /westpp
      /front-end
      /wrapper
        /atom3
        /westpp
    /win32-msvc6.0
      /back-end
        /atom3 back_end
        /westpp back_end
      /front-end
      /wrapper
        /atom3 wrapper
        /westpp wrapper
  /doc
  /etc
    /linux
  /include
  /lib
    /linux
    /win-msvc6.0
  /src
    /atom3
    /back-end
      /atom3
        /cpp
        /py
      /westpp
    /front-end
    /MSLE
    /wrapper
      /atom3
      /westpp
  /test
```

Figuur 4.9: Directory-structuur

4.4.2 Gentle

De lexicale en grammaticale specificaties werden door Prof. Hans Vangheluwe in Gentle geprogrammeerd.

Gentle [4] is een geïntegreerd systeem om compilers te schrijven. Gentle laat toe een compiler te definiëren op een hoog niveau zodat implementatiedetails verborgen blijven. Daarvoor wordt gebruik gemaakt van de *Gentle Compiler Description Language*. Doordat Gentle op een hoger niveau staat is er geen controle over de datastructuren die gebruikt worden en de opbouw van de abstract syntax tree. Om die reden wordt de Gentle specificatie van de MSL-compiler omgezet naar een flex- en bison-specificatie.

4.4.3 Front-end

De fornt-end bevat, zoals in hoofdstuk I besproken werd, een lexicale en een grammaticale analyse. De specificaties werden respectievelijk in flex en bison geschreven.

Lexicale analyse

De specificatie van de lexicale regels werd oorspronkelijk gedaan in Gentle en werd omgezet naar een flex-specificatie.

De lexicale analyse herkent onder andere identifiers, integers, reals en strings. De lexicale analyse kan geneste commentaren herkennen. Zo is het mogelijk de volgende commentaar te laten verwerken door de lexicale analyse:

```
/*
 commentaar
/* geneste commentaar */
// geneste lijncommentaar
*/
```

De volledige lexicale flex-specificatie is te vinden in appendix C, pagina (97). Er wordt ook verwezen naar 3.1.3 voor meer informatie.

Grammaticale analyse

De grammaticale productieregels in Gentle zijn omgezet naar hun tegenhangers in bison. De bison code is te vinden in appendix D (pagina 107). Indien de parser, door middel van wrapper geïmplementeerd in `src/wrapper/atom3`, wordt opgeroepen vanuit Python, wordt tijdens de grammaticale analyse een abstract syntax tree opgebouwd als een Python-object.

4.4.4 Abstract Syntax Tree

Om de herbruikbaarheid van de flex en bison code te verhogen is er een abstracte basisklasse, `class MSLUObjectClass`, gedefinieerd. De abstracte basisklasse vormt een interface voor het opbouwen van de abstract syntax tree. De functieoproepen, voor het construeren van de abstract syntax tree, in de bison-code zijn enkel oproepen naar één functie, `MSLUbuildObject`. De functie heeft de vorm:

```
MSLUOPTR
MSLUbuildObject (enum AllObjectTypes vtype,
                 int lline,
                 int lcol,
                 void *ptr1 = NULL,
                 void *ptr2 = NULL,
                 void *ptr3 = NULL,
                 void *ptr4 = NULL);
```

Hierin is `vtype` het type object dat gecreëerd moet worden of waar een operatie op dient gedaan te worden, bijvoorbeeld het toevoegen van een element aan een lijst. `lline` en `lcol` verwijzen respectievelijk naar het lijn- en kolomnummer in het MSL-USER bestand. `ptr1` tot `ptr4` zijn pointers naar symbolen van de rechterleden van een productie. Ze worden gebruikt om nieuwe objecten op te bouwen of bestaande te wijzigen. De volledige definitie van de interface is terug te vinden vinden in appendix E op pagina 138.

Voor het gebruik van de compiler in Python werd van de abstracte basisklasse `class MSLUObjectClass` een klasse `PyMSLUObjectClass` afgeleid. Deze klasse heeft een attribuut `ParseObject` van het type `PyObject*`. Dit object, zal nadat het parsen succesvol beëindigd is, de abstract syntax tree voorstellen.

Om de abstracte syntax tree in Python op te bouwen is een Python-klasse `class toplevelObject` geïmplementeerd waarvan analoge klassen als diegene gedefinieerd in de interface zijn afgeleid. Er is dus een overeenstemming tussen de interfaceklassen en de Python-klassen.

4.4.5 Back-end

Bij gebruik van de parser als een Python-extensie is er na de lexicale en de grammaticale analyse een abstract syntax tree ter beschikking van de back-end. De abstract syntax tree is een Python-object. De back-end functionaliteit van de MSL compiler werd om die reden in Python geschreven. De back-end is in staat MSL-USER code te verwerken en voor een beperkt aantal modellen MSL-EXEC code te produceren. De Python-methoden die daarvoor gebruikt worden zijn respectievelijk `printMSLE(self)` en `printMSLU (self, file)`.

4.4.6 Verloop van het compilatieproces

Dit deel handelt over het gebruik van de compiler als Python-extensie. Hoe de installatie in zijn werk gaat is te vinden in 4.4.8.

Het verloop van het compilatieproces is voorgesteld in figuur 4.10. De verschillende stappen worden hierna besproken:

1. **WEST++ model** : De eerste stap is het beschrijven van het model.
2. **MSL-USER** : Het model wordt omgezet naar MSL-USER.
3. **flex** : De lexicale specificaties worden door flex omgezet naar een uitvoerbaar programma dat een lexicale analyse uitvoert. Dit programma voert de lexicale analyse uit en geeft als output tokens aan de volgende fase.
4. **bison** : De grammaticale specificaties worden door bison omgezet naar een uitvoerbaar programma dat een grammaticale analyse uitvoert. Dit programma heeft als input de tokens afkomstig van de lexicale analyse en bouwt de abstract syntax tree (AST) op.

5. **AST** : De bekomen abstract syntax tree kan dan onmiddellijk gebruikt worden door de code generatie of dit kan via AToM^3 gebeuren.

6. **code gen.** : Uit de bekomen abstract syntax tree wordt MSL-USER of MSL-EXEC gegenereerd.

4.4.7 Voorbeeld

Het voorbeeld zal de verschillende stappen uit de sectie 4.4.6 doorlopen.

In de biologie wordt vaak de populatiegrootte van een organisme gesimuleerd. Als voorbeeld nemen we een dergelijke populatiesimulatie: een *roofdier-prooisysteem* (*predator-prey system*). Een roofdier-prooisysteem bestaat uit twee populaties:

Roofdierpopulatie : De leden van de *roofdierpopulatie* consumeren die uit de prooi populatie, bijvoorbeeld katten.

Prooipopulatie : De leden van de *prooipopulatie* worden door de roofdierpopulatie geconsumeerd, bijvoorbeeld muizen.

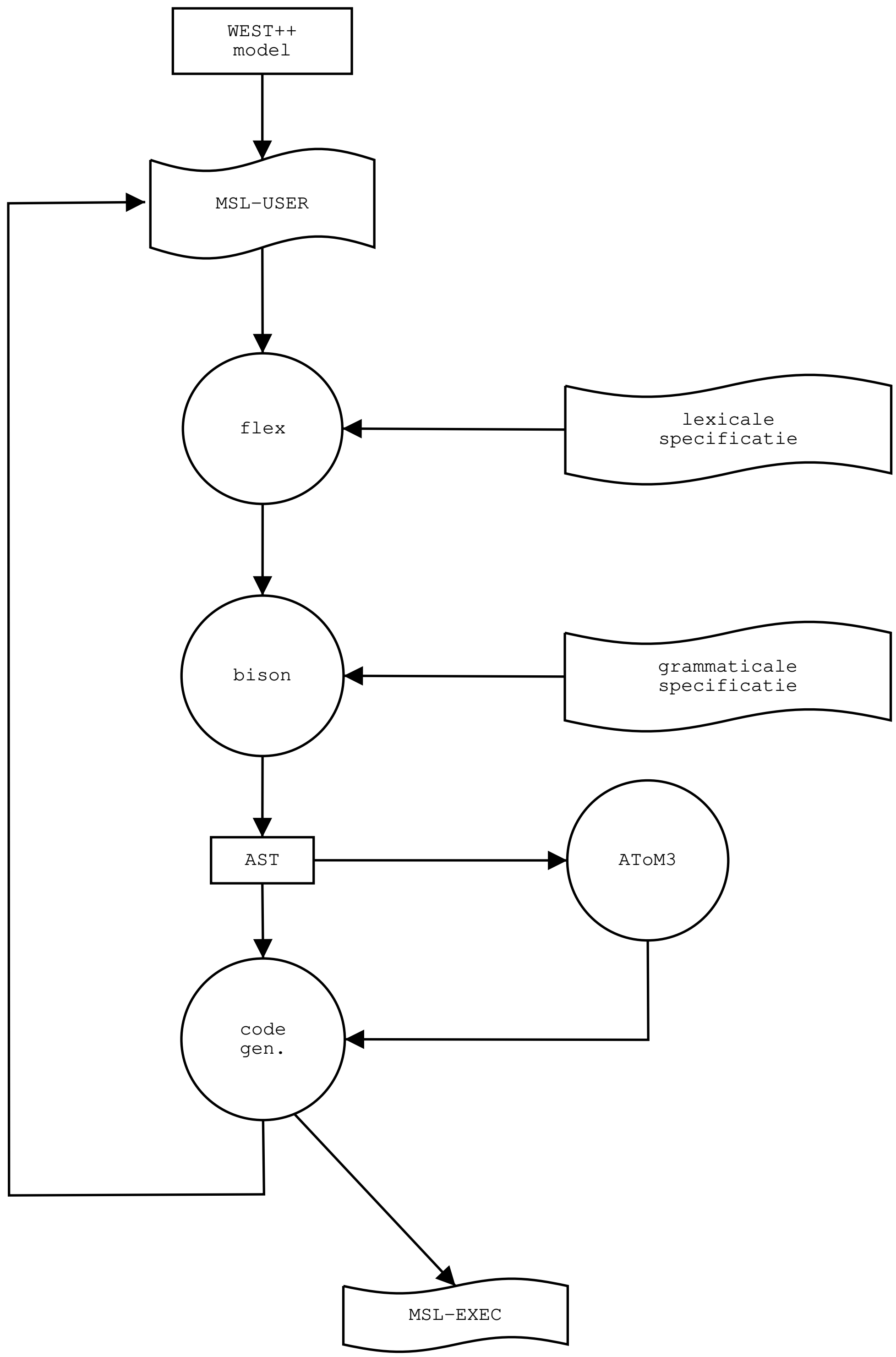
De differentiaalvergelijkingen die het model beschrijven zijn (*(*Lotka-Volterra vergelijkingen):

$$\frac{dP_{pred}}{dt} = -aP_{pred} + kbP_{pred}P_{prey}$$
$$\frac{dP_{prey}}{dt} = cP_{prey} - bP_{pred}P_{prey}$$

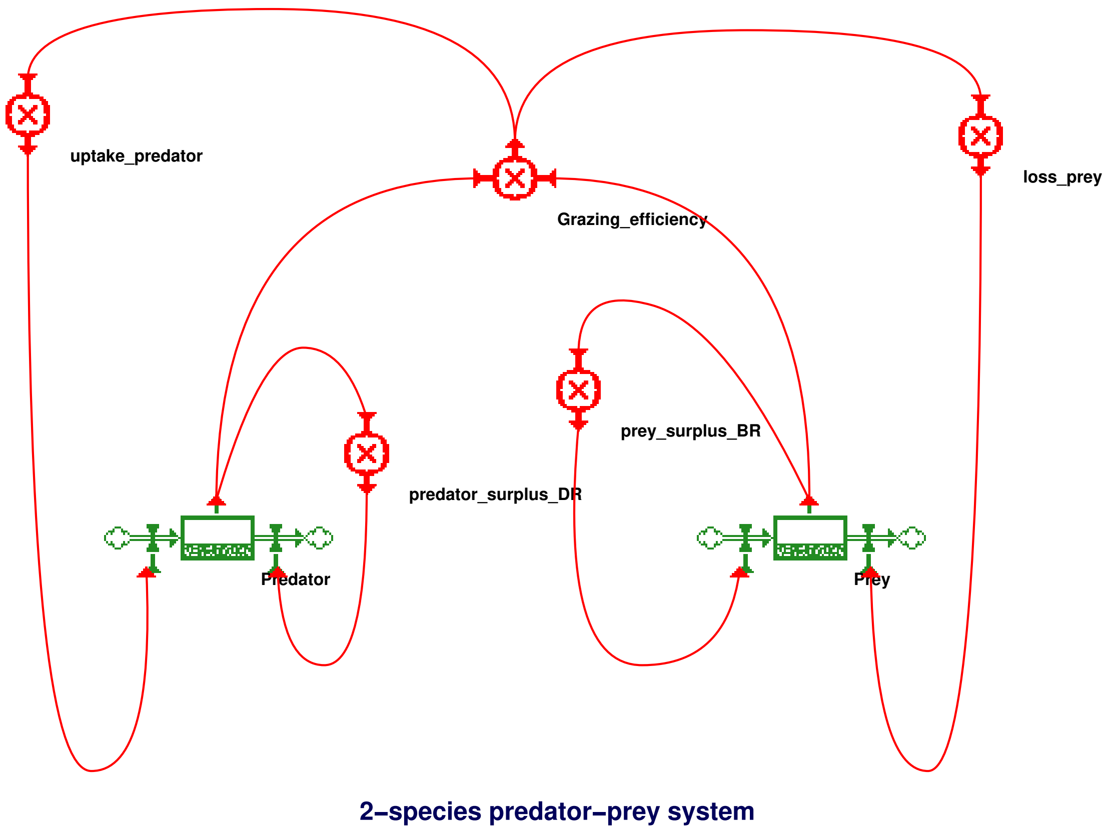
Met dP_{pred} de grootte van de roofdierpopulatie, dP_{prey} de grootte van de prooipopulatie, a de sterfcoëfficiënt van de roofdieren, b de sterfcoëfficiënt van de prooien, c de geboortecoëfficiënt van de prooien en k de efficiëntiefactor.

De eerste stap bestaat uit het modelleren van het systeem in WEST++. Dit roofdier-prooisysteem kan aan de hand van een *Forrester System Dynamics* model in WEST++ gemodelleerd worden (figuur 4.11).

WEST++ produceert MSL-USER 3.1. Het model zal om die reden met de hand beschreven worden als een MSL-USER 4.0 model. De MSL-USER 4.0 code van het roofdier-prooisysteem is:



Figuur 4.10: MSL compiler flow chart



Figuur 4.11: Roofdier-prooimodel in WEST++

```
// Model of Predator-prey system

OBJ PredatorPrey "Predator-prey system" : Generic :=
{
   // The declarations

   OBJ b "death rate prey": real := 0.05,
   OBJ k "efficiency" : real := 0.2,
   OBJ a "death rate predator": real := 0.8,
   OBJ c "growth rate prey" : real := 3.2,
   OBJ t : IndepVar,
   OBJ pa "predator population" : PhysicalQuantityType := 10,
   OBJ ps "prey population" : PhysicalQuantityType := 100,

   DERIV(pa, [t,]) = - a * pa + k * b * pa * ps,
   DERIV(ps, [t,]) = c * ps - b * ps * pa,
}
```

In dit model hebben de verschillende parameters een waarde gekregen.
Het bestand dat dit model bevat wordt voor verwerking aan de compiler aangeboden. De compiler zal een lexicale en een grammaticale analyse uitvoeren. Eens dit achter de rug is, is er een abstract syntax tree opgebouwd. Deze abstract syntax tree kan in AToM^3 gebruikt worden om MSL-USER of MSL-EXEC naar een bestand te schrijven.

De bekomen MSL-EXEC code zal,indien er gevraagd wordt MSL-EXEC te genereren, bestaan uit een ".c" en een ".h" bestand.

Het bestand PredatorPrey.h:

```
class PredatorPreyClass : public MSLEDAEModelClass
{
 public:
 PredatorPreyClass(StringType name_arg);
 void ComputeOutput(void);
 void ComputeInitial(void);
 void ComputeState(void);
 void ComputeTerminal(void);
};
```

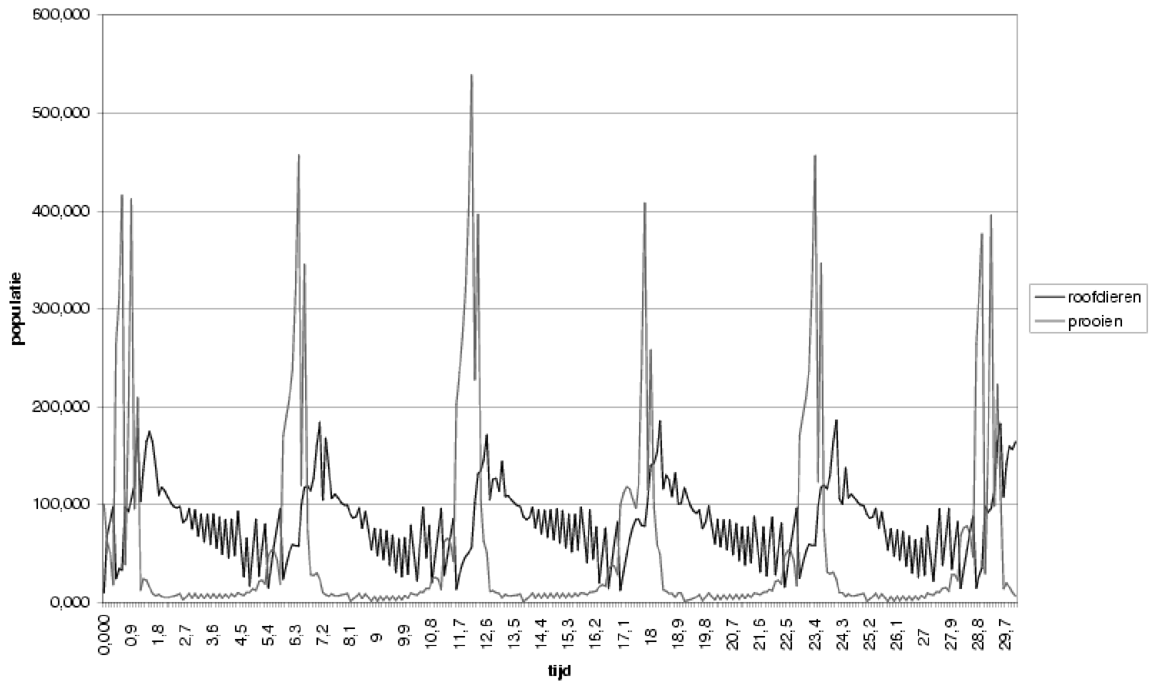
Het bestand PredatorPrey.cpp:

```
#include "MSLE.h"
#include "PredatorPrey.h"

#define _b_ ParamValues[0]
#define _k_ ParamValues[1]
#define _a_ ParamValues[2]
#define _c_ ParamValues[3]
#define _t_ IndepVarValues[0]
#define _pa_ DerStateVarValues[0]
#define _ps_ DerStateVarValues[1]
#define _D_pa_ Derivatives[0]
#define _D_ps_ Derivatives[1]

PredatorPreyClass :: PredatorPreyClass(StringType name_arg)
{
 set_name(name_arg);
 set_description("Predator-prey system");
 set_class_name("PredatorPreyClass");

 set_no_params(4);
 set_no_param_values(4);
 set_param(0, new MSLEParamClass("b", "", 0.05));
 set_param(1, new MSLEParamClass("k", "", 0.2));
 set_param(2, new MSLEParamClass("a", "", 0.8));
 set_param(3, new MSLEParamClass("c", "", 3.2));

 set_no_indep_vars(1);
 set_no_indep_var_values(1);
 set_indep_var(0, new MSLEIndepVarClass("t", "s"));

 set_no_output_vars(2);
 set_no_output_var_values(2);
 set_output_var(0, new MSLEOutputVarClass("pa_out", "", 0));
 set_output_var(1, new MSLEOutputVarClass("ps_out", "", 0));

 set_no_der_state_vars(2);
 set_no_der_state_var_values(2);
 set_der_state_var(0, new MSLEDerStateVarClass("pa", "", 10));
```

```
 set_der_state_var(1, new MSLEDerStateVarClass("ps", "", 100));

 Link();
 Reset();
}

void PredatorPreyClass :: ComputeOutput(void)
{
}

void PredatorPreyClass :: ComputeInitial(void)
{
}

void PredatorPreyClass :: ComputeState(void)
{
 _D_pa_ = − _a_ * _pa_ + _k_ * _b_ * _pa_ * _ps_ ;
 _D_ps_ = _c_ * _ps_ − _b_ * _ps_ * _pa_ ;
}

void PredatorPreyClass :: ComputeTerminal(void)
{
}

#undef _b_
#undef _a_
#undef _k_
#undef _c_
#undef _t_
#undef _pa_
#undef _ps_
```

Deze bestanden kunnen dan gebruikt worden om een simulatie uit te voeren in WEST++. De plot van de simulatie is opgenomen in figuur 4.12. De lijn met de hoge pieken is de roofdierpopulatie en de lijn met kleine pieken is de prooipopulatie.

Uit de figuur is af te leiden dat de prooipopulatie eerst stijgt omdat er slechts weinig roofdieren zijn. Door het stijgend aantal prooien is er meer voedsel voor de roofdieren, wat leidt tot een stijging van hun aantal. Op een bepaald moment is het aantal roofdieren zo



Figuur 4.12: Predator-prey plot

groot dat hun consumptie van prooien groter is dan het ritme waarmee deze zich voortplanten. Hierdoor zal het aantal prooien dalen. Door het dalend aantal prooien is er minder voedsel beschikbaar voor de roofdieren en begint hun aantal eveneens te dalen. Op het moment dat de consumptie van prooien kleiner is dan hun geboortecoëffiënt begint de populatiegrootte van de prooien weer te stijgen.

4.4.8 Installatie

In deze sectie wordt beschreven hoe alles geïnstalleerd moet worden op een computer. In de distributie staat dit in het bestand `INSTALL`.

WIN32

MSLU relies on the following software components, which are NOT part of the distribution:

- flex
- bison
- Microsoft Visual Studio 6.0

In order to install MSLU on your Win32 platform, follow the following steps:

- Unpack the MSLU distribution in an appropriate directory.
- Open the workspace-file in `<install-dir>/build/win32-msvc6.0/`
- Build the projects:
    1. atom3 wrapper
    2. westpp wrapper
- When the linker reports an error that it can not find the library '`python22_d.lib`', do the following:

  NOTE: This section only applies to source distributions of Python prior to September 7, 2002. After that, any source distribution has a fix that makes this no longer necessary. (Source after that date also fixes another Python bug, so updating is highly recommended.)

If you're trying to compile this module with a default Python installation, you'll need to make a couple modifications. Locate pyconfig.h in your Python include directory, and search for 'python22_d.lib' (Somewhere around line 336) ... It should look somewhat like this:

```
#ifdef _DEBUG
#pragma comment(lib, "python22_d.lib")
#else
#pragma comment(lib, "python22.lib")
#endif
```

Change 'python22_d.lib' to 'python22.lib':

```
#ifdef _DEBUG
#pragma comment(lib, "python22.lib")
#else
#pragma comment(lib, "python22.lib")
#endif
```

A few lines down, you'll want to comment out the line that says '#define Py_DEBUG'.

```
#ifdef _DEBUG
// #define Py_DEBUG
#endif
```

Save the changes.

- To build the documentation:

  1. Set the input directory of Doxygen to `$MSLU_ROOT_PATH/src` in the file `$MSLU_ROOT_PATH/etc/Doxyfile` .
  2. Set the output directory of Doxygen to `$MSLU_ROOT_PATH/doc` in the file `$MSLU_ROOT_PATH/etc/Doxyfile` .
  3. Run Doxygen with `$MSLU_ROOT_PATH/etc/Doxyfile` as configuration file.

LINUX

MSLU relies on the following software components, which are NOT part of the distribution:

- GNU flex version 2.5.4
- GNU Bison 1.35
- gcc version 3.2
- g++ version 3.2
- Python 2.2

In order to install MSLU on your Linux platform, follow the following steps:

- Use the shell `/bin/sh` .
- Unpack the MSLU distribution in an appropriate directory.
- Set the following environment variables: `MSLU_ROOT_PATH` (where the installation is)
- Change variables in `$MSLU_ROOT_PATH/etc/linux/MSLU.conf.sh if` necessary.
- Execute, using 'source', the appropriate build environment configuration script. It can be found at `$MSLU_ROOT_PATH/etc/linux/MSLU.conf.sh` .
- Invoke make in `$MSLU_ROOT_PATH` .

So basically do this:

1. `export MSLU_ROOT_PATH=<...>`
2. `source $MSLU_ROOT_PATH/etc/linux/MSLU.conf.sh`
3. `cd $MSLU_ROOT_PATH`
4. `make`

You can run a test like this:

```
make test
```

To build the documentation:

1. Set the input directory of Doxygen to `$MSLU_ROOT_PATH/src` in the file `$MSLU_ROOT_PATH/etc/Doxyfile` .
2. Set the output directory of Doxygen to `$MSLU_ROOT_PATH/doc` in the file `$MSLU_ROOT_PATH/etc/Doxyfile` .
3. Run Doxygen with `$MSLU_ROOT_PATH/etc/Doxyfile` as configuration file.

AToM^3

It is possible to use the module `MSL_atom3_wrapper` in AToM^3 or Python. The parser is called like this: `MSLUparser(<file_object>)` .

To use it in AToM^3 first read: http://moncs.cs.mcgill.ca/people/mprovost/tutorial_b/tut_a_main.html .

- Copy the files in `$MSLU_ROOT_PATH/src/back-end/atom3/py/`, `$MSLU_ROOT_PATH/src/atom3/` and the file `$MSLU_ROOT_PATH/bin/<platform>/MSLU_atom3_wrapper.*` to your desired AToM^3 directory.
- Set up a button to invoke the parser containing the action: `execfile("<your_AToM3_dir>/MSLUparsebutton.py")` .
- Set up a button to invoke the MSLU generator containing the action: `execfile("<your_AToM3_dir>/MSLUprintbutton.py")` .
- Set up a button to invoke the MSLE generator containing the action: `execfile("<your_AToM3_dir>/MSLEprintbutton.py")` .

4.5 Besluit

De opbouw van de MSL compiler gebeurt door gebruik te maken van verschillende prototypes. Specifieke problemen van bepaalde onderdelen van de compiler worden opgelost per prototype. Op die manier blijft het bos door de bomen zichtbaar.

Er wordt gebruik gemaakt van flex en bison om de lexicale en grammaticale specificatie van de twee prototypes en de MSL compiler te implementeren.

De parser kan vanuit Python of AToM^3 worden gebruikt omdat rond de compiler een wrapper gelegd wordt. Deze wrapper is een Python-extensie. De opbouw van de abstract syntax tree gebeurt in Python met behulp van de Python-wrapper. Uitgaande van de abstract syntax tree kan terug MSL-USER of MSL-EXEC gegenereerd worden.

Hoofdstuk V

Algemeen besluit

In deze scriptie werd een compiler gebouwd die de MSL-USER 4.0 taal kan verwerken. De MSL compiler is integreerbaar met AToM^3. De doelstelling, een stelsel van vergelijkingen verwerken en hiervoor MSL-EXEC genereren, werd gedeeltelijk bereikt en kan op eenvoudige modellen worden toegepast. Een uitgebreidere functionaliteit werd niet gecodeerd omdat daarvoor de tijd te beperkt was.
De softwarecrisis steekt weer de kop op.

Om het doel te bereiken werd eerst een introductie gegeven over compilers. Het compilatieproces kan worden opgedeeld in zes fasen: de lexicale analyse, de grammaticale analyse, de semantische analyse, de intermediaire code generatie, de code optimalisatie en de code generatie. Een compiler bestaat uit twee delen: de front-end (lexicale analyse, grammaticale analyse, semantische analyse en intermediaire code generatie) en de back-end (code optimalisatie en code generatie). Het strikt gescheiden houden van de front-end en de back-end verhoogt de herbruikbaarheid van de twee componenten. Naast deze twee delen is er nog de symbol table en de abstract syntax tree. De symbol table interageert met alle fasen van het compilatieproces en hoort principieel niet thuis in de front- of back-end. De opbouw start tijdens de lexicale analyse en wordt tijdens de volgende fasen aangepast en bevraagd.
Lexicale analyse doet een spellingscontrole op de invoer. Tijdens de grammaticale analyse wordt gecontroleerd of de compiler input voldoet aan de grammatica van de taal. De semantische analyse kijkt of hetgeen in het invoerbestand staat allemaal wel een betekenis heeft.
Het resultaat van deze drie fasen is een abstract syntax tree. De abstract syntax tree staat

dan ter beschikking van de intermediaire code generatie of, indien er geen intermediaire code generatie is, onmiddellijk van de code generatie met eventueel code optimalisatie. De geproduceerde code is het eindproduct van de compiler.

De compiler dient geïntegreerd te worden met AToM^3. AToM^3 is een modelleer-omgeving die gebruik maakt van meta-modellen om een nieuwe modelleer-omgeving te creëren. De taal gebruikt om AToM^3 te ontwikkelen is Python. Python is een hoge niveau object-georiënteerde taal. In Python kunnen extensies gebruikt worden die het mogelijk maken functionaliteit in een andere taal geschreven te gebruiken. Dit mechanisme kan uitgebreid worden naar een callback systeem. In dergelijk callback mechanisme wordt vanuit de niet-Python-code Python-functionaliteit opgeroepen. Dergelijk systeem wordt gebruikt om de MSL parser vanuit AToM^3 op te roepen. De MSL parser bouwt dan de abstract syntax tree op in Python zodat deze voor AToM^3 toegankelijk is.

De MSL compiler dient gebruikt te worden in WEST++. WEST++ is een modelleer- en simulatie-omgeving die vooral gebruikt wordt voor het modelleren en simuleren van waterzuivering. Het programma bestaat uit twee hoofdcomponenten: een modelleer-omgeving en een experimenteer-omgeving. Deze zijn van elkaar gescheiden om de modulariteit te verhogen. De modellen worden grafisch of tekstueel beschreven. De grafische modellen worden vertaald naar MSL-USER. Handmatig geschreven modellen worden onmiddellijk in MSL-USER gecodeerd.

MSL-USER 4.0 is een objectgeoriënteerde taal die toelaat om de dynamica van een model op een declaratieve manier te beschrijven.

Eens een model volledig gespecificeerd is in MSL-USER wordt het door de parser omgezet naar MSL-EXEC. De MSL-EXEC code kan vervolgens gecompileerd worden en verschillende soorten virtuele experimenten kunnen dan uitgevoerd worden binnen de WEST++ omgeving.

De opbouw van de MSL compiler gebeurt door gebruik te maken van verschillende prototypes. Specifieke problemen van bepaalde onderdelen van de compiler worden opgelost per prototype. Op die manier blijft het bos door de bomen zichtbaar.

De tools flex en bison worden aangewend om de lexicale en grammaticale specificatie van de twee prototype compilers en de uiteindelijke MSL compiler te specificeren en te genereren.

De parser kan vanuit Python of AToM^3 worden gebruikt als een extensie die beroep doet op een callback mechanisme. De opbouw van de abstract syntax tree vindt op die manier plaats in Python. Door gebruik te maken van de abstract syntax tree kan terug MSL-USER of MSL-EXEC gegenereerd worden.

Bijlage A

MSLU 4.0 syntax

```
<MSL file> ::= <topObject>
             |
<topObject> ::= <object>
<object> ::= <explicitObject>
           | <implicitObject>
<explicitObject> ::= <objectdeclaration> := <object>
                   | <objectdeclaration>
<objectdeclaration> ::= OBJ <identifier> <optionalComment> : <type>
<implicitObject> ::= <typedeclaration>
                   | <equationObject>
                   | <expressionObject>
<typedeclaration> ::= TYPE <type>
 | TYPE <identifier> <optionalComment> = <type> <optionalDefaultObject>
<optionalDefaultObject> ::= DEFAULT <object>
                          |
<type> ::= <basicType>
         | <setType>
         | <productType>
         | <vectorType>
         | <operatorType>
         | <unionType>
         | <referenceType>
```

```
         | <bracketedType>
         | <identifierType>
<basicType> ::= Bottom
              | Generic
              | TypeType
              | Integer
              | Real
              | Character
              | String
              | <enumeratedType>
<enumeratedType> ::= ENUM <identifierlist> END_ENUM
<identifierlist ::= <identifierlist> , <identifier>
                  | <identifier>
<setType> ::= SET <type> END_SET
<productType> ::= PRODUCT <typelist> END_PRODUCT
<vectorType> ::= <orientation> VECTOR <optionalRange> OF <type> END_VECTOR
<orientation> ::= ROW
                | COLUMN
                |
<optionalRange> ::= <expressionObject>
                  | <expressionObject> .. <expressionObject>
                  |
<operatorType> ::= OPERATOR <optionalType> -> <optionalType> END_OPERATOR
<unionType> ::= UNION <typelist> END_UNION
<referenceType> ::= REF <type> END_REF
<bracketedType> ::= ( <type> )
<identifierType> ::= <identifier>
<optionalType> ::= <type>
                 |
<typelist> ::= <typelist> , <type>
             | <type>
<equationObject> ::= <expressionObject> <equalORassign> <expressionObject>
<equalORassign> ::= =
                  | <->
```

```
                  | =:
                  | -->
                  | :=
                  | <--
<expressionObject> ::= <logical_OR_expressionObject>
<logical_OR_expressionObject> ::= <logical_AND_expressionObject>
 | <logical_OR_expressionObject> || <logical_AND_expressionObject>
<logical_AND_expressionObject> ::= <equality_expressionObject>
 | <logical_AND_expressionObject> && <equality_expressionObject>
<equality_expressionObject> ::= <relational_expressionObject>
 | <equality_expressionObject> == <relational_expressionObject>
 | <equality_expressionObject> != <relational_expressionObject>
 | <equality_expressionObject> <> <relational_expressionObject>
<relational_expressionObject> ::= <additive_expressionObject>
 | <relational_expressionObject> < <additive_expressionObject>
 | <relational_expressionObject> <= <additive_expressionObject>
 | <relational_expressionObject> > <additive_expressionObject>
 | <relational_expressionObject> >= <additive_expressionObject>
<additive_expressionObject> ::= <mult_expressionObject>
                              | <additive_expressionObject> + <mult_expressionObject>
                              | <additive_expressionObject> - <mult_expressionObject>
<mult_expressionObject> ::= <unary_expressionObject>
                          | <mult_expressionObject> * <unary_expressionObject>
                          | <mult_expressionObject> / <unary_expressionObject>
                          | <mult_expressionObject> MOD <unary_expressionObject>
<unary_expressionObject> ::= <vector_selector_expressionObject>
                           | - <vector_selector_expressionObject>
                           | + <vector_selector_expressionObject>
                           | NOT <vector_selector_expressionObject>
<vector_selector_expressionObject> ::= <record_selector_expressionObject>
 | <vector_selector_expressionObject> [ <expressionObject> ]
<record_selector_expressionObject> ::= <actualObject>
 | <record_selector_expressionObject> . <actualObject>
<actualObject> ::= <nullObject>
```

```
                 | <integerNumberObject>
                 | <realNumberObject>
                 | <characterObject>
                 | <stringObject>
                 | <setObject>
                 | <productObject>
                 | <vectorObject>
                 | <operatorObject>
                 | <evalOperatorObject>
                 | <enumORidentifierObject>
                 | <bracketedObject>
<nullObject> ::= NULL
<integerNumberObject> ::= <IntegerNumber>
<realNumberObject> ::= <RealNumber>
<characterObject> ::= <Character>
<stringObject> ::= <String>
<setObject> ::= { <objectlist> }
<productObject> ::= (: <rowobjectlist> :)
<vectorObject> ::= [ <rowORcolumnObjectlist> ]
<operatorObject> ::= (: <object> -> <object> :) WITH <setORvectorObject>
<setORvectorObject> ::= <setObject>
                      | <vectorObject>
<evalOperatorObject> ::= <identifier> ( <objectlist> )
<enumORidentifierObject> ::= <identifier>
<bracketedObject> ::= ( <implicitObject> )
<objectlist> ::= <rowobjectlist>
               | <object>
               |
<rowORcolumnObjectlist> ::= <rowobjectlist>
                          | <columnobjectlist>
<rowobjectlist> ::= <objectORnull> , <restOFrowobjectlist>
<restOFrowobjectlist> ::= <restOFrowobjectlist> , <objectORnull>
                        | <objectORnull>
<columnobjectlist> ::= <objectORnull> ; <restOFcolumnobjectlist>
```

```
<restOFcolumnobjectlist> ::= <restOFcolumnobjectlist> ; <objectORnull>
                           | <objectORnull>
<objectORnull> ::= <object>
                 |
<identifier> ::= <Ident>
<optionalComment> ::= <String>
                    |
```

Bijlage B

Yacc code prototype 1

```
%{

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "node.h"

unsigned node_nr = 1;
extern FILE *yyin, *yyout;
FILE *file_out;
int lineno = 1;
struct node *start_node_ptr;
extern enum type;
const int X_START = 35, Y_START = 35;
int i = 1;
int alfa = 2;
int node_size; // grootte van een node horizontaal or verticaal
char *direction;
char expr[200];
%}

%start START

%token PLUS MUL INV NUMBER

%union
```

```
{
        int number;
        struct node *node_ptr;
}

%token <number> NUMBER
%type <node_ptr> expression
%type <node_ptr> term
%type <node_ptr> factor
%type <node_ptr> number
%type <node_ptr> START

%%

START:      expression {
                         start_node_ptr = $1;
                       }
;

expression:  expression  PLUS term {
                                     $$ = init_op_ptr (T_PLUS, node_nr++, $1, $3);
                                     if (strcmp(direction,"v") == 0 || strcmp(direction,"V") == 0)
                                       (*$$).x_coor = ((*$1).x_coor + (*$3).x_coor) / alfa;
                                     else
                                       (*$$).y_coor = ((*$1).y_coor + (*$3).y_coor) / alfa;
                                   }

          |  term {
                    $$ = $1;
                  }
   ;

term:        term MUL factor {
                                $$ = init_op_ptr (T_MUL, node_nr++, $1, $3);
                                if (strcmp(direction,"v") == 0 || strcmp(direction,"V") == 0)
                                  (*$$).x_coor = ((*$1).x_coor + (*$3).x_coor) / alfa;
                                else
                                  (*$$).y_coor = ((*$1).y_coor + (*$3).y_coor) / alfa;
                             }

          |  factor {
```

```
                    $$ = $1;
                   }
   ;

factor:     '(' expression ')' { $$ = $2;}

        |   INV number {
                        (*$2).node_type = T_INV;
                        $$ = $2;
                        ++node_nr; // nodig omdat een negatief getal slechts 1 knoop is in de parse tree
                                   // maar 2 in de AToM3 tree. zie ook het printen van een T_INV knoop
                      }
        |   number {
                     $$ = $1;
                   }
   ;

number:     NUMBER {
                     $$ = init_number_ptr($1, node_nr++);
                     if (strcmp(direction,"v") == 0 || strcmp(direction,"V") == 0)
                       (*$$).x_coor = node_size*i;
                     else
                       (*$$).y_coor = node_size*i;
                     i++;

                   }
   ;


%%

main(int argc, char **argv)
{

 char *fileName;
 int test_input;


  strcpy(fileName,argv[1]);
  strcat(fileName, ".py");
```

```
file_out = fopen(fileName, "w");
if (!file_out){
  fprintf(stderr,"could not create outout file\n");
  exit(1);
}

test_input = fscanf(stdin, "%d", &alfa);*/
alfa = 2;

/* horizontale of verticale boom? */


fprintf(stdout, "Horizontal [h] or vertical [v] tree? ");

direction = (char*) malloc(5 * sizeof(char));
if (direction == NULL) {
  fprintf(stderr, "Out of memory.\n");
  exit(1);
}

fscanf(stdin, "%s", direction);
while((strcmp(direction,"h") != 0 && strcmp(direction,"H") != 0 && strcmp(direction,"v") != 0 && strcmp(direction,"V") != 0) ||
       strlen(direction) > 5){
  fprintf(stdout, "Wrong input\n");
  fprintf(stdout, "Horizontal [h] or vertical [v] tree? ");
  free(direction);
  direction = NULL;
  direction = (char*) malloc(5 * sizeof(char));
  fprintf(stdout,"after malloc\n");
  if (direction == NULL) {
    fprintf(stderr, "Out of memory.\n");
    exit(1);
  }
  fscanf(stdin, "%s", direction);
}

printf("Expression? \n");
fscanf(stdin, "%s", expr);
printf("expr = %s\n", expr);

if(strcmp(direction,"h") == 0 || strcmp(direction,"H") == 0) {
```

```
  node_size = 50;
}
else {
  node_size = 70;
}

yyout = file_out;

fprintf(yyout, "# File created by MSL-parser prototype1_3\n");
fprintf(yyout, "# Do not edit unless you know what you are doing!\n\n");

fprintf(yyout, "from graph_ASG_ERmetaMetaModel import *\n");
fprintf(yyout, "from stickylink import *\n");
fprintf(yyout, "from widthXfillXdecoration import *\n");
fprintf(yyout, "from ASG_algebra_entities import *\n");
fprintf(yyout, "from alg_operator import *\n");
fprintf(yyout, "from constant import *\n");
fprintf(yyout, "from operates_on import *\n");
fprintf(yyout, "from ATOM3BottomType import *\n");
fprintf(yyout, "from ATOM3String import *\n");
fprintf(yyout, "from ATOM3Constraint import *\n");
fprintf(yyout, "from ATOM3Attribute import *\n");
fprintf(yyout, "from ATOM3Enum import *\n");
fprintf(yyout, "from ATOM3Appearance import *\n");
fprintf(yyout, "from ATOM3Float import *\n");
fprintf(yyout, "from ATOM3Connection import *\n");
fprintf(yyout, "from ATOM3Boolean import *\n");
fprintf(yyout, "from ATOM3Link import *\n");
fprintf(yyout, "from ATOM3Integer import *\n");
fprintf(yyout, "from ATOM3List import *\n");
fprintf(yyout, "from ATOM3Port import *\n");
fprintf(yyout, "\n\n");

fprintf(yyout, "def %s(self, rootNode):\n\n",argv[1]);

printf("start parsing\n");
yyparse();
printf("parsing done\n");

if(strcmp(direction,"v") == 0 || strcmp(direction,"V") == 0) {
  (*start_node_ptr).y_coor = Y_START;
```

```
  }
  else {
    (*start_node_ptr).x_coor = X_START;
  }

  set_coordinates(start_node_ptr, direction);

  print_node(yyout, start_node_ptr, direction);

  fprintf(yyout, "newfunction = %s\n\n", argv[1]);
  fprintf(yyout, "loadedMMName = 'algebra_entities'\n");
  fprintf(stdout, "File %s has been generated.\n", fileName);

  // file sluiten en geheugen vrijgeven

  free_tree(start_node_ptr);
  if (!fclose(yyin)) fprintf(stdout, "input file closed\n");
  if (!fclose(yyout)) fprintf(stdout, "output file closed\n");

}
```

Bijlage C

Lexicale specificatie in flex van MSLU (MSLU.l)

```
%{

#include <string.h>
#include <stdlib.h>
#include "strings.h"
#include "errmsg.h"
#include "MSLUobject.h"
#include "y.tab.h"

extern YYSTYPE yylval;
extern long yypos;

#define yysetpos() {yylval.attr[0] = yypos; yypos += yyleng; }

extern char *expr;

void SkipRestOfComment (void);
void SkipRestOfLineComment (void);
void ReadStringLiteral (void);

void AppendToString(char);

%}
%%
```

```
";" { yysetpos(); return SEMICOLON; }
"WITH" { yysetpos(); return WITH; }
":)" { yysetpos(); return COLON_RPAREN; }
"(:" { yysetpos(); return LPAREN_COLON; }
"}" { yysetpos(); return RBRACE; }
"{" { yysetpos(); return LBRACE; }
"." { yysetpos(); return DOT; }
"]" { yysetpos(); return RBRACKET; }
"[" { yysetpos(); return LBRACKET; }
"NOT" { yysetpos(); return NOT; }
"MOD" { yysetpos(); return MOD; }
"/" { yysetpos(); return SLASH; }
"*" { yysetpos(); return ASTERISK; }
"-" { yysetpos(); return MINUS; }
"+" { yysetpos(); return PLUS; }
">=" { yysetpos(); return GREATER_EQUAL; }
">" { yysetpos(); return GREATER; }
"<=" { yysetpos(); return LESS_EQUAL; }
"<" { yysetpos(); return LESS; }
"<>" { yysetpos(); return LESS_GREATER; }
"!=" { yysetpos(); return EXCLAM_EQUAL; }
"==" { yysetpos(); return EQUAL_EQUAL; }
"&&" { yysetpos(); return AMPERSAND_AMPERSAND; }
"||" { yysetpos(); return BAR_BAR; }
"<--" { yysetpos(); return LESS_MINUS_MINUS; }
"-->" { yysetpos(); return MINUS_MINUS_GREATER; }
"=:" { yysetpos(); return EQUAL_COLON; }
"<->" { yysetpos(); return LESS_MINUS_GREATER; }
")" { yysetpos(); return RPAREN; }
"(" { yysetpos(); return LPAREN; }
"END_REF" { yysetpos(); return END_UNDERSCORE_REF; }
"REF" { yysetpos(); return REF; }
"END_UNION" { yysetpos(); return END_UNDERSCORE_UNION; }
"UNION" { yysetpos(); return UNION; }
"END_OPERATOR" { yysetpos(); return END_UNDERSCORE_OPERATOR; }
"->" { yysetpos(); return MINUS_GREATER; }
"OPERATOR" { yysetpos(); return OPERATOR; }
".." { yysetpos(); return DOT_DOT; }
"COLUMN" { yysetpos(); return COLUMN; }
"ROW" { yysetpos(); return ROW; }
"END_VECTOR" { yysetpos(); return END_UNDERSCORE_VECTOR; }
```

```
"OF" { yysetpos(); return OF; }
"VECTOR" { yysetpos(); return VECTOR; }
"END_PRODUCT" { yysetpos(); return END_UNDERSCORE_PRODUCT; }
"PRODUCT" { yysetpos(); return PRODUCT; }
"END_SET" { yysetpos(); return END_UNDERSCORE_SET; }
"SET" { yysetpos(); return SET; }
"," { yysetpos(); return COMMA; }
"END_ENUM" { yysetpos(); return END_UNDERSCORE_ENUM; }
"ENUM" { yysetpos(); return ENUM; }
"String" { yysetpos(); return String_kw; }
"Character" { yysetpos(); return Character_kw; }
"Real" { yysetpos(); return Real; }
"Integer" { yysetpos(); return Integer; }
"TypeType" { yysetpos(); return TypeType; }
"Generic" { yysetpos(); return Generic; }
"Bottom" { yysetpos(); return Bottom; }
"DEFAULT" { yysetpos(); return DEFAULT; }
"=" { yysetpos(); return EQUAL; }
"TYPE" { yysetpos(); return TYPE; }
":" { yysetpos(); return COLON; }
"OBJ" { yysetpos(); return OBJ; }
":=" { yysetpos(); return COLON_EQUAL; }

"NULL" {
 /* the NULL token of MSLU */

 yysetpos();
 return Null;
}

[0-9]+ {
 /* a positive integer constant                          */
 /* note how an arbitrary number of leading 0's is allowed */

 yylval.attr[1] = atoi(yytext);
 yysetpos();
 return IntegerNumber;
}

[0-9]+"."[0-9]+([eE][-+]?[0-9]+)? {
 /* A positive Real constant */
```

```
 /* Implemented as a C double */

 double *dp;
 dp = (double *)malloc(sizeof(double));
 *dp = atof(yytext);
 yylval.attr[1] = (long)dp; /* a pointer to the Real constant */
 yysetpos();
 return RealNumber;
}

'\\.' {
 /* An escaped character      */
 /* The meaning is as in ANSI C */

 unsigned char c;

 switch (yytext[2])
 {
  case 'n': c = '\n'; break;   /* newline                        */
  case 't': c = '\t'; break;   /* tab                            */
  case 'v': c = '\v'; break;   /* vertical tab                   */
  case 'a': c = '\a'; break;   /* audible alert (BEL)            */
  case 'b': c = '\b'; break;   /* backspace                      */
  case 'r': c = '\r'; break;   /* carriage return                */
  case 'f': c = '\f'; break;   /* formfeed                       */
  case '0': c = '\0'; break;   /* NULL byte (terminates C strings) */
  default : c = yytext[2]; break; /* other characters stand for  */
                               /* themselves when escaped        */
 }
 yylval.attr[1] = (long)c;
 yysetpos();
 return Character;
}

'[^']' {
 /* A normal character is returned as is */
 yylval.attr[1] = (long)yytext[1];
 yysetpos();
 return Character;
}
```

```
'\\[0-3][0-7][0-7]' {
 /* character given as an octal number */

 long l = 0;
 int i;

 for (i = 2; i <= 4; i++)
 {
  l = l * 8 + (yytext[i] - '0');
 }
 yylval.attr[1] = (long)l;
 yysetpos();
 return Character;
}

'\\x[0-9a-fA-F][0-9a-fA-F]' {
 /* might want to allow \X as well as \x in the future */
 /* character given as a hexadecimal number   */

 long l = 0;
 long ladd;
 int i;
 unsigned char c;

 for (i = 2; i <= 3; i++)
 {
  c = yytext[i];

  if (('a' <= c) && (c <= 'f'))
   ladd = c - 'a' + 10; // 'a' hex has value 10 ...
  else if (('A' <= c) && (c <= 'F'))
   ladd = c - 'A' + 10; // 'A' hex has value 10 ...
  else // if (('0' <= c) && (c <= '9'))
   ladd = c - '0';    // '0' has a hex value of 0 ...

  l = l*16 + ladd;
 }
 yylval.attr[1] = (long)l;
 yysetpos();
 return Character;
}
```

```
\" {
 /* a C-style string literal */

 yypos++;          /* initial '"' */
 ReadStringLiteral(); /* reading the rest of the string and handling */
                   /* of e.g., EOF inside a string         */
 return String;
}

[A-Za-z_][A-Za-z0-9_]* {
 /* identifiers start with an alpha character or _             */
 /* followed by a possibly empty sequence of alphanumerics or _ */

 char *id;
 id = (char*) malloc(sizeof(char)*yyleng);
 id = yytext;
 yylval.attr[1] = (long) id;
 return Ident;
}

"/*" {
 /* comments to be put anywhere whitespace can be put */
 /* SkipRestOfComment() handles the */
 /* inside of the comments. This includes nested  */
 /* comments and EOF inside a comment.           */
 yypos++; yypos++; /* initial \/\* */
 SkipRestOfComment();
}

"//" {
 /* comments, skipped till the end of the line        */
 /* SkipRestOfLineComment(); handles */
 /* the skipping                                      */
 yypos++; yypos++; /* initial '//' */
 SkipRestOfLineComment();
}

\  { yypos += 1; }
\n { yyPosToNextLine(); }
\t { yypos += 1; }
```

```
. { yysetpos(); yyerror("illegal token"); }
%%
/*                                                */
/*                                                */
/* Functions to handle C and C++ style comments */
/* once (f)lex has detected we're                 */
/* inside comments                                */
/*                                                */
/* This file implements:                          */
/*                                                */
/* void SkipRestOfComment(void)                   */
/* Handles text inside C-style comments           */
/* Nesting of comments is allowed                 */
/* EOF reached before end of comment is detected */
/*                                                */
/* void SkipRestOfLineComment(void)               */
/* Handles text inside C++-style comments         */
/* In this case, the rest of the line is eaten    */
/*                                                */
/* void ReadStringLiteral(void)                   */
/* Handles string literals                        */
/* These are used in MSL for string objects as well */
/* as for OBJ or TYPE declaration comments       */
/* Special cases handled:                         */
/* - EOF inside string (forgot to terminate) or  */
/*   forgot to escape " inside a string          */
/* - Multi-line strings, leaving \n inside string */
/* - \-escaped characters                         */
/*                                                */

#define TRUE 1
#define EOFCHAR EOF

#ifdef __cplusplus
#define NEXTCHAR (yypos++, yyinput())
#else
#define NEXTCHAR (yypos++, input())
#endif

/* When inside a C-style comment, nesting is allowed */
```

```
/* EOF reached before end of comment is detected */
void SkipRestOfComment(void)
{
 int ch;        /* next read character */
 int nest = 1; /* nesting counter */

 ch = NEXTCHAR;
 while (TRUE)
 {
  switch (ch) {
   case '\n':           /* ignore newline
    yyPosToNextLine (); * but increment the line counter */
    ch = NEXTCHAR;
    break;
   case EOFCHAR :  /* EOF inside commment */
    yyerror ("EOF found before end of comment\
            \n>>> probably forgot to close the comment");
    exit (1);
    break;
   case '*':            /* possible end of comment */
    ch = NEXTCHAR; /* look ahead one more character */
    if (ch == '/')
    {
     nest--;
     if (nest == 0)    /* reached end of outermost comment */
      { return; }
     ch = NEXTCHAR;
    }
    break;
   case '/':            /* possible start of comment */
    ch = NEXTCHAR; /* look ahead one more character */
    if (ch == '*')
    {
     nest++;
     ch = NEXTCHAR;
    }
    break;
   default:             /* eat character */
    ch = NEXTCHAR;
  }
 }
```

```
}

/* When inside a C++-style comment, the rest of a line is eaten */
void SkipRestOfLineComment(void)
{
 int ch;                    /* next read character */

 do                  /* eat characters until EOLN */
 {
  ch = NEXTCHAR;
 } while (ch != '\n');

 yyPosToNextLine();  /* increment the line counter */
}

/* String literals are used in MSL for string objects */
/* as well as for OBJ or TYPE declaration comments */
/* Handles:                                          */
/*  EOF inside string (forgot to terminate) or forgot */
/*  to escape " inside a string                      */
/*  Multi-line strings, leaving \n inside string     */
/*  \-escaped characters                             */
void ReadStringLiteral(void)
{
 int ch; /* next read character */

 while (TRUE)
 {
  ch = NEXTCHAR;
  switch (ch)
  {
   case EOFCHAR :
    yyerror("EOF occurred inside a string\
          \n>>> probably forgot to terminate the string\
          \n>>> or forgot to escape \\\" a \" inside a string");
    break;
   case '\n' :
    yyPosToNextLine (); /* multi-line string, advance line counter */
    AppendToString ((char) ch); /* leave the '\n' in the string */
    break;
   case '\\' :          /* a '\' followed by a character means an */
```

```
                         /* escaped character. We leave them in as is */
    AppendToString((char) ch);
    ch = NEXTCHAR;
    AppendToString ((char) ch);
    break;
   case '"' :            /* end of string reached */
    yylval.attr[0] = yypos;
    GetStringRef((char**)&yylval.attr[1]); // pointer to string
    return;
    break;
   default :
    AppendToString((char) ch); /* just add the character to the string */
  }
 }
}

#ifndef yywrap
int yywrap() { return 1; }
#endif
```

Bijlage D

Grammaticale specificatie in bison van MSLU (MSLU.y)

```
%{
#include <string>
#include <fstream>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include "errmsg.h"
#include "MSLUobject.h"

using namespace std;

extern int yylex();
MSLUOPTR MSLUParsedObject;

%}

%union
{
  long attr[2];
  MSLUOPTR MSLUObj;
  int ival;
  double dval;
  char *strval;
}
```

```
%start ROOT_
%token <attr> Null 257
%token <attr> IntegerNumber 258
%token <attr> RealNumber 259
%token <attr> Character 260
%token <attr> String 261
%token <attr> Ident 262
%token SEMICOLON 263
%token WITH 264
%token COLON_RPAREN 265
%token LPAREN_COLON 266
%token RBRACE 267
%token LBRACE 268
%token DOT 269
%token RBRACKET 270
%token LBRACKET 271
%token NOT 272
%token MOD 273
%token SLASH 274
%token ASTERISK 275
%token MINUS 276
%token PLUS 277
%token GREATER_EQUAL 278
%token GREATER 279
%token LESS_EQUAL 280
%token LESS 281
%token LESS_GREATER 282
%token EXCLAM_EQUAL 283
%token EQUAL_EQUAL 284
%token AMPERSAND_AMPERSAND 285
%token BAR_BAR 286
%token LESS_MINUS_MINUS 287
%token MINUS_MINUS_GREATER 288
%token EQUAL_COLON 289
%token LESS_MINUS_GREATER 290
%token RPAREN 291
%token LPAREN 292
%token END_UNDERSCORE_REF 293
%token REF 294
%token END_UNDERSCORE_UNION 295
```

```
%token UNION 296
%token END_UNDERSCORE_OPERATOR 297
%token MINUS_GREATER 298
%token OPERATOR 299
%token DOT_DOT 300
%token COLUMN 301
%token ROW 302
%token END_UNDERSCORE_VECTOR 303
%token OF 304
%token VECTOR 305
%token END_UNDERSCORE_PRODUCT 306
%token PRODUCT 307
%token END_UNDERSCORE_SET 308
%token SET 309
%token COMMA 310
%token END_UNDERSCORE_ENUM 311
%token ENUM 312
%token String_kw 313
%token Character_kw 314
%token Real 315
%token Integer 316
%token TypeType 317
%token Generic 318
%token Bottom 319
%token DEFAULT 320
%token EQUAL 321
%token TYPE 322
%token COLON 323
%token OBJ 324
%token COLON_EQUAL 325

%type <MSLUObj> ROOT_ toplevel topObject object explicitObject implicitObject
objectdeclaration identifier typedeclaration equationObject
expressionObject type optionalDefaultObject basicType setType productType
vectorType operatorType unionType referenceType bracketedType identifierType
enumeratedType identifierlist typelist optionalRange optionalType
logical_OR_expressionObject
logical_AND_expressionObject equality_expressionObject relational_expressionObject
additive_expressionObject mult_expressionObject unary_expressionObject
vector_selector_expressionObject record_selector_expressionObject actualObject
nullObject integerNumberObject realNumberObject characterObject stringObject
```

```
setObject productObject vectorObject operatorObject evalOperatorObject
enumORidentifierObject bracketedObject objectORnull objectlist rowobjectlist
rowORcolumnObjectlist setORvectorObject vector_selector_expressionObject
columnobjectlist restOFrowobjectlist restOFcolumnobjectlist optionalComment

%type <strval> orientation equalORassign


%%

ROOT_:
   openoutput
   toplevel
   closeoutput

   {
     MSLUParsedObject = $2;
   }
   ;
openoutput:

   {
     cout << "MSL translation starting" << endl;
   }
   ;
toplevel:
   topObject

   {
      $$ = MSLUbuildObject (TOP_LEVEL_OBJECT, yyLine(), yyColumn(), $1);
   }
   ;

topObject:
   object

   {
     $$ = MSLUbuildObject (TOP_OBJECT, yyLine(), yyColumn(), $1);
   }

   |
```

```
    {
      $$ = MSLUbuildObject (TOP_OBJECT, yyLine(), yyColumn());
    }
    ;

closeoutput:

   {
     cout << "MSL-USER-4.0 translation ended successfully" << endl;
   }
   ;

object:
    explicitObject

    {
      $$ = $1;
    }

    |
    implicitObject

    {
      $$ = $1;
    }
    ;

explicitObject:
    objectdeclaration
    COLON_EQUAL
    object

    {
      $$ = MSLUbuildObject (EXPLICIT_OBJECT, yyLine(), yyColumn(), $1, $3);
    }

    |
    objectdeclaration

    {
      $$ = $1;
```

```
    }
    ;

objectdeclaration:
    OBJ
    identifier
    optionalComment
    COLON
    type

    {
      $$ = MSLUbuildObject (OBJECT_DECLARATION_OBJECT, yyLine(), yyColumn(), $2, $3, $5);
    }
    ;

implicitObject:
    typedeclaration

    {
      $$ = $1;
    }

    |
    equationObject

    {
      $$ = $1;
    }

    |
    expressionObject

    {
      $$ = $1;
    }
    ;

typedeclaration:
    TYPE
    type
```

```
    {
      $$ = MSLUbuildObject (TYPE_DECLARATION_OBJECT, yyLine(), yyColumn(), $2);
    }

    |
    TYPE
    identifier
    optionalComment
    EQUAL
    type
    optionalDefaultObject

    {
      $$ = MSLUbuildObject (TYPE_DECLARATION_OBJECT, yyLine(), yyColumn(), $2, $3, $5, $6);
    }
    ;

optionalDefaultObject:
    DEFAULT
    object

    {
      $$ = MSLUbuildObject (OPTIONAL_DEFAULT_OBJECT, yyLine(), yyColumn(), $2);
    }

    |

    {
      $$ = MSLUbuildObject (OPTIONAL_DEFAULT_OBJECT, yyLine(), yyColumn());
    }
    ;

type:
    basicType

    {
       $$ = $1;
    }


    |
```

```
setType

{
    $$ = $1;
}

|
productType

{
    $$ = $1;
}

|
vectorType

{
    $$ = $1;
}

|
operatorType

{
    $$ = $1;
}

|
unionType

{
  $$ = $1;
}

|
referenceType

{
    $$ = $1;
}
```

```
    |
    bracketedType

    {
      $$ = $1;
    }

    |
    identifierType

    {
       $$ = $1;
    }
    ;

basicType:
    Bottom

    {
      $$ = MSLUbuildObject (BOTTOM_TYPE_OBJECT, yyLine(), yyColumn());
    }

    |
    Generic

    {
      $$ = MSLUbuildObject (GENERIC_TYPE_OBJECT, yyLine(), yyColumn());
    }
    |

    TypeType

    {
      $$ = MSLUbuildObject (TYPE_TYPE_TYPE_OBJECT, yyLine(), yyColumn());
    }

    |
    Integer

    {
      $$ = MSLUbuildObject (INTEGER_TYPE_OBJECT, yyLine(), yyColumn());
```

```
    }

    |
    Real

    {
      $$ = MSLUbuildObject (REAL_TYPE_OBJECT, yyLine(), yyColumn());
    }

    |
    Character_kw

    {
      $$ = MSLUbuildObject (CHARACTER_KW_TYPE_OBJECT, yyLine(), yyColumn());
    }

    |
    String_kw

    {
      $$ = MSLUbuildObject (STRING_KW_TYPE_OBJECT, yyLine(), yyColumn());
    }

    |
    enumeratedType

    {
      $$ = $1;
    }
    ;

enumeratedType:
    ENUM
    identifierlist
    END_UNDERSCORE_ENUM

    {
       $$ = MSLUbuildObject (ENUMERATED_TYPE_OBJECT, yyLine(), yyColumn(), $2);
    }
    ;
```

```
identifierlist:
    identifierlist
    COMMA
    identifier

    {
        $$ = MSLUbuildObject (IDENTIFIER_LIST_OBJECT, yyLine(), yyColumn(), $1, $3);
    }

    |
    identifier
    {
        $$ = MSLUbuildObject (IDENTIFIER_LIST_OBJECT, yyLine(), yyColumn(), $1);
    }
    ;

setType:
    SET
    type
    END_UNDERSCORE_SET

    {
        $$ = MSLUbuildObject (SET_TYPE_OBJECT, yyLine(), yyColumn(), $2);
    }
    ;

productType:
    PRODUCT
    typelist
    END_UNDERSCORE_PRODUCT

    {
        $$ = MSLUbuildObject (PRODUCT_TYPE_OBJECT, yyLine(), yyColumn(), $2);
    }
    ;

vectorType:
    orientation
    VECTOR
    optionalRange
    OF
```

```
    type
    END_UNDERSCORE_VECTOR

    {
      $$ = MSLUbuildObject (VECTOR_TYPE_OBJECT, yyLine(), yyColumn(), $1, $3, $5);
    }
    ;
orientation:
    ROW

    {
     $$ = (char*)"ROW";
    }

    |
    COLUMN

    {
     $$ = (char*)"COLUMN";
    }

    |

    {
      $$ = (char*)"";
    }
    ;

optionalRange:
    expressionObject

    {
      $$ = $1;
    }

    |
    expressionObject
    DOT_DOT
    expressionObject

    {
```

```
      $$ = MSLUbuildObject (OPTIONAL_RANGE_OBJECT, yyLine(), yyColumn(), $1, $3);
   }

   |

   {
      $$ = MSLUbuildObject (OPTIONAL_RANGE_OBJECT, yyLine(), yyColumn());
   }
   ;

operatorType:
   OPERATOR
   optionalType
   MINUS_GREATER
   optionalType
   END_UNDERSCORE_OPERATOR

   {
      $$ = MSLUbuildObject (OPERATOR_TYPE_OBJECT, yyLine(), yyColumn(), $2, $4);
   }
   ;

unionType:
   UNION
   typelist
   END_UNDERSCORE_UNION

   {
     $$ = MSLUbuildObject (UNION_TYPE_OBJECT, yyLine(), yyColumn(), $2);
   }
   ;

referenceType:
   REF
   type
   END_UNDERSCORE_REF

   {
      $$ = MSLUbuildObject (REFERENCE_TYPE_OBJECT, yyLine(), yyColumn(), $2);
   }
   ;
```

```
bracketedType:
    LPAREN
    type
    RPAREN

    {
        $$ = MSLUbuildObject (BRACKETED_TYPE_OBJECT, yyLine(), yyColumn(), $2);
    }
    ;

identifierType:
    identifier

    {
        $$ = $1;
    }
    ;

optionalType:
    type

    {
        $$ = $1;
    }

    |

    {
        $$ = MSLUbuildObject (OPTIONAL_TYPE_OBJECT, yyLine(), yyColumn());
    }
    ;

typelist:
    typelist
    COMMA
    type

    {
        $$ = MSLUbuildObject (TYPE_LIST_OBJECT, yyLine(), yyColumn(), $1, $3);
    }
```

```
|
type
{
   $$ = MSLUbuildObject (TYPE_LIST_OBJECT, yyLine(), yyColumn(), $1);
}
;

equationObject:
   expressionObject
   equalORassign
   expressionObject

   {
      $$ = MSLUbuildObject (EQUATION_OBJECT, yyLine(), yyColumn(), $1, $2, $3);
   }
   ;

equalORassign:
   EQUAL

   {
      $$ = (char*)"="
   }

   |
   LESS_MINUS_GREATER

   {
      $$ = (char*)"<->"
   }

   |
   EQUAL_COLON

   {
      $$ = (char*)"=:"
   }

   |
   MINUS_MINUS_GREATER
```

```
    {
        $$ = (char*)"-->"
    }

    |
    COLON_EQUAL

    {
        $$ = (char*)":="
    }

    |
    LESS_MINUS_MINUS

    {
        $$ = (char*)"<--"
    }
    ;

expressionObject:
    logical_OR_expressionObject

    {
        $$ = $1;
    }
    ;

logical_OR_expressionObject:
    logical_AND_expressionObject

    {
        $$ = $1;
    }

    |
    logical_OR_expressionObject
    BAR_BAR
    logical_AND_expressionObject

    {
```

```
    $$ = MSLUbuildObject (LOGICAL_OR_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, $3);
  }
  ;

logical_AND_expressionObject:
  equality_expressionObject

  {
    $$ = $1;
  }

  |
  logical_AND_expressionObject
  AMPERSAND_AMPERSAND
  equality_expressionObject

  {
    $$ = MSLUbuildObject (LOGICAL_AND_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, $3);
  }
  ;

equality_expressionObject:
  relational_expressionObject

  {
    $$ = $1;
  }

  |
  equality_expressionObject
  EQUAL_EQUAL
  relational_expressionObject

  {
    $$ =
    MSLUbuildObject (EQUALITY_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"==", $3);
  }

  |
  equality_expressionObject
  EXCLAM_EQUAL
```

```
relational_expressionObject

{
  $$ =
  MSLUbuildObject (EQUALITY_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"!=", $3);
}

|
equality_expressionObject
LESS_GREATER
relational_expressionObject

{
  $$ =
  MSLUbuildObject (EQUALITY_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"<>", $3);
}
;

relational_expressionObject:
additive_expressionObject

{
   $$ = $1;
}

|
relational_expressionObject
LESS
additive_expressionObject

{
  $$ =
  MSLUbuildObject (RELATIONAL_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"<", $3);
}

|
relational_expressionObject
LESS_EQUAL
additive_expressionObject

{
```

```
    $$ =
    MSLUbuildObject (RELATIONAL_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"<=", $3);
  }

  |
  relational_expressionObject
  GREATER
  additive_expressionObject

  {
    $$ =
    MSLUbuildObject (RELATIONAL_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)">", $3);
  }

  |
  relational_expressionObject
  GREATER_EQUAL
  additive_expressionObject

  {
    $$ =
    MSLUbuildObject (RELATIONAL_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)">=", $3);
  }
  ;

additive_expressionObject:
  mult_expressionObject

  {
    $$ = $1;
  }

  |
  additive_expressionObject
  PLUS
  mult_expressionObject

  {
    $$ =
    MSLUbuildObject (ADDITIVE_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"+", $3);
  }
```

```
    |
    additive_expressionObject
    MINUS
    mult_expressionObject

    {
      $$ =
      MSLUbuildObject (ADDITIVE_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"-", $3);
    }
    ;

mult_expressionObject:
    unary_expressionObject

    {
      $$ = $1;
    }

    |
    mult_expressionObject
    ASTERISK
    unary_expressionObject

    {
      $$ = MSLUbuildObject (MULT_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"*", $3);
    }

    |
    mult_expressionObject
    SLASH
    unary_expressionObject

    {
      $$ = MSLUbuildObject (MULT_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"/", $3);
    }

    |
    mult_expressionObject
    MOD
    unary_expressionObject
```

```
  {
    $$ = MSLUbuildObject (MULT_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, (char*)"MOD", $3);
  }
  ;

unary_expressionObject:
  vector_selector_expressionObject

  {
     $$ = $1;
  }

  |
  MINUS
  vector_selector_expressionObject

  {
     $$ = MSLUbuildObject (UNARY_EXPRESSION_OBJECT, yyLine(), yyColumn(), (char*)"-", $2);
  }

  |
  PLUS
  vector_selector_expressionObject

  {
     $$ = MSLUbuildObject (UNARY_EXPRESSION_OBJECT, yyLine(), yyColumn(), (char*)"+", $2);
  }

  |
  NOT
  vector_selector_expressionObject

  {
     $$ = MSLUbuildObject (UNARY_EXPRESSION_OBJECT, yyLine(), yyColumn(), (char*)"NOT", $2);
  }
  ;

vector_selector_expressionObject:
  record_selector_expressionObject
```

```
    {
      $$ = $1;
    }

    |
    vector_selector_expressionObject
    LBRACKET
    expressionObject
    RBRACKET

    {
      $$ =
      MSLUbuildObject (VECTOR_SELECTOR_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, $3);
    }
    ;

record_selector_expressionObject:
    actualObject

    {
       $$ = $1;
    }

    |
    record_selector_expressionObject
    DOT
    actualObject

    {
      $$ =
      MSLUbuildObject (RECORD_SELECTOR_EXPRESSION_OBJECT, yyLine(), yyColumn(), $1, $3);
    }
    ;

actualObject:
    nullObject

    {
      $$ = $1;
    }
```

```
|
integerNumberObject

{
  $$ = $1;
}

|
realNumberObject

{
  $$ = $1;
}

|
characterObject

{
  $$ = $1;
}

|
stringObject

{
  $$ = $1;
}

|
setObject

{
  $$ = $1;
}

|
productObject

{
  $$ = $1;
}
```

```
    |
    vectorObject

    {
      $$ = $1;
    }

    |
    operatorObject

    {
      $$ = $1;
    }

    |
    evalOperatorObject

    {
      $$ = $1;
    }

    |
    enumORidentifierObject

    {
      $$ = $1;
    }

    |
    bracketedObject

    {
      $$ = $1;
    }
    ;

nullObject:
    Null

    {
```

```
      $$ = MSLUbuildObject (NULL_OBJECT, yyLine(), yyColumn());
    }
    ;

integerNumberObject:
    IntegerNumber

    {
      $$ = MSLUbuildObject (INTEGER_NUMBER_OBJECT, yyLine(), yyColumn(), &$1[1]);
    }
    ;

realNumberObject:
    RealNumber

    {
       $$ = MSLUbuildObject (REAL_NUMBER_OBJECT, yyLine(), yyColumn(), (double*)$1[1]);
    }
    ;

characterObject:
    Character

    {
       $$ = MSLUbuildObject (CHARACTER_OBJECT, yyLine(), yyColumn(), &$1[1]);
    }
    ;

stringObject:
    String

    {
       $$ = MSLUbuildObject (STRING_OBJECT, yyLine(), yyColumn(), (char*) $1[1]);
    }
    ;

setObject:
    LBRACE
    objectlist
    RBRACE
```

```
  {
    $$ = MSLUbuildObject (SET_OBJECT, yyLine(), yyColumn(), $2);
  }
  ;

productObject:
  LPAREN_COLON
  rowobjectlist
  COLON_RPAREN

  {
    $$ = MSLUbuildObject (PRODUCT_OBJECT, yyLine(), yyColumn(), $2);
  }
  ;

vectorObject:
  LBRACKET
  rowORcolumnObjectlist
  RBRACKET

  {
    $$ = MSLUbuildObject (VECTOR_OBJECT, yyLine(), yyColumn(), $2);
  }
  ;

operatorObject:
  LPAREN_COLON
  object
  MINUS_GREATER
  object
  COLON_RPAREN
  WITH
  setORvectorObject

  {
     $$ = MSLUbuildObject (OPERATOR_OBJECT, yyLine(), yyColumn(), $2, $4, $7);
  }
  ;

setORvectorObject:
  setObject
```

```
    {
      $$ = $1;
    }

    |
    vectorObject

    {
     $$ = $1;
    }
    ;

evalOperatorObject:
    identifier
    LPAREN
    objectlist
    RPAREN

    {
      $$ = MSLUbuildObject (EVAL_OPERATOR_OBJECT, yyLine(), yyColumn(), $1, $3);
    }
    ;

enumORidentifierObject:
    identifier

    {
     $$ = $1;
    }
    ;

bracketedObject:
    LPAREN
    implicitObject
    RPAREN
    {
     $$ = MSLUbuildObject (BRACKETED_OBJECT, yyLine(), yyColumn(), $2);
    }
    ;
```

```
objectlist:
    rowobjectlist

    {
        $$ = $1;
    }

    |
    object

    {
        $$ = MSLUbuildObject (OBJECT_LIST_OBJECT, yyLine(), yyColumn(), $1);
    }

    |

    {
        $$ = MSLUbuildObject (OBJECT_LIST_OBJECT, yyLine(), yyColumn());
    }
    ;

rowORcolumnObjectlist:
    rowobjectlist

    {
        $$ = $1;
    }

    |
    columnobjectlist

    {
        $$ = $1;
    }
    ;

rowobjectlist:
    objectORnull
    COMMA
    restOFrowobjectlist
```

```
   {
      $$ = MSLUbuildObject (ROW_OBJECT_LIST_OBJECT, yyLine(), yyColumn(), $1, $3);
   }
   ;

restOFrowobjectlist:
   restOFrowobjectlist
   COMMA
   objectORnull

   {
      $$ = MSLUbuildObject (REST_OF_ROW_OBJECT_LIST_OBJECT, yyLine(), yyColumn(), $1, $3);
   }

   |
   objectORnull

   {
     $$ = MSLUbuildObject (REST_OF_ROW_OBJECT_LIST_OBJECT, yyLine(), yyColumn(), $1);
   }
   ;

columnobjectlist:
   objectORnull
   SEMICOLON
   restOFcolumnobjectlist

   {
      $$ = MSLUbuildObject (COLUMN_OBJECT_LIST_OBJECT, yyLine(), yyColumn(), $1, $3);
   }
   ;

restOFcolumnobjectlist:
   restOFcolumnobjectlist
   SEMICOLON
   objectORnull

   {
     $$ =
     MSLUbuildObject (REST_OF_COLUMN_OBJECT_LIST_OBJECT, yyLine(), yyColumn(), $1, $3);
   }
```

```
    |
    objectORnull

    {
      $$ = MSLUbuildObject (REST_OF_COLUMN_OBJECT_LIST_OBJECT, yyLine(), yyColumn(), $1);
    }
    ;

objectORnull:
    object

    {
     $$ = $1;
    }

    |

    {
      $$ = MSLUbuildObject (OBJECT_OR_NULL_OBJECT, yyLine(), yyColumn());
    }
    ;

identifier:
    Ident

    {
      $$ = MSLUbuildObject (IDENTIFIER_OBJECT, yyLine(), yyColumn(), (char*) $1[1]);
    }
    ;

optionalComment:
    String

    {
      $$ = MSLUbuildObject (OPTIONAL_COMMENT_OBJECT, yyLine(), yyColumn(), (char*) $1[1]);
    }

    |

    {
```

```
        $$ = MSLUbuildObject (OPTIONAL_COMMENT_OBJECT, yyLine(), yyColumn(), (char*)"");
    }
    ;

%%
```

Bijlage E

Interface MSLUobject.h

```
#ifndef MSLU_OBJECT_H
#define MSLU_OBJECT_H

#include <string>

using namespace std;

#define MSLUOPTR MSLUObjectClass*

/// Enum
/** enum of MSLU objects */
enum AllObjectTypes
{
  TOP_LEVEL_OBJECT,
  TOP_OBJECT,
  EXPLICIT_OBJECT,
  OBJECT_DECLARATION_OBJECT,
  TYPE_DECLARATION_OBJECT,
  OPTIONAL_DEFAULT_OBJECT,
  BOTTOM_TYPE_OBJECT,
  GENERIC_TYPE_OBJECT,
  TYPE_TYPE_TYPE_OBJECT,
  INTEGER_TYPE_OBJECT,
  REAL_TYPE_OBJECT,
  CHARACTER_KW_TYPE_OBJECT,
  STRING_KW_TYPE_OBJECT,
  ENUMERATED_TYPE_OBJECT,
```

IDENTIFIER_LIST_OBJECT,
SET_TYPE_OBJECT,
PRODUCT_TYPE_OBJECT,
VECTOR_TYPE_OBJECT,
OPTIONAL_RANGE_OBJECT,
OPERATOR_TYPE_OBJECT,
UNION_TYPE_OBJECT,
REFERENCE_TYPE_OBJECT,
BRACKETED_TYPE_OBJECT,
IDENTIFIER_TYPE_OBJECT,
OPTIONAL_TYPE_OBJECT,
TYPE_LIST_OBJECT,
EQUATION_OBJECT,
LOGICAL_OR_EXPRESSION_OBJECT,
LOGICAL_AND_EXPRESSION_OBJECT,
EQUALITY_EXPRESSION_OBJECT,
RELATIONAL_EXPRESSION_OBJECT,
ADDITIVE_EXPRESSION_OBJECT,
MULT_EXPRESSION_OBJECT,
UNARY_EXPRESSION_OBJECT,
VECTOR_SELECTOR_EXPRESSION_OBJECT,
RECORD_SELECTOR_EXPRESSION_OBJECT,
NULL_OBJECT,
INTEGER_NUMBER_OBJECT,
REAL_NUMBER_OBJECT,
CHARACTER_OBJECT,
STRING_OBJECT,
SET_OBJECT,
PRODUCT_OBJECT,
VECTOR_OBJECT,
OPERATOR_OBJECT,
EVAL_OPERATOR_OBJECT,
ENUM_OR_IDENTIFIER_OBJECT,
BRACKETED_OBJECT,
OBJECT_LIST_OBJECT,
ROW_OBJECT_LIST_OBJECT,
REST_OF_ROW_OBJECT_LIST_OBJECT,
COLUMN_OBJECT_LIST_OBJECT,
REST_OF_COLUMN_OBJECT_LIST_OBJECT,
OBJECT_OR_NULL_OBJECT,
IDENTIFIER_OBJECT,

```
  OPTIONAL_COMMENT_OBJECT
};


/// MSLUObject class
/** A base class for MSLU */

class MSLUObjectClass
{
  /// First line
  int first_line;
  /// First column
  int first_column;
  /// Last line
  int last_line;
  /// Last column
  int last_column;
public:
  void SetLocation(int lastLine, int lastColumn);
  int GetFirstLine();
  int GetFirstColumn();
  int GetLastLine();
  int GetLastColumn();
};

/// Identifier Class
class IdentifierClass:public MSLUObjectClass
{};

/// Optional Comment Class
class OptionalCommentClass:public MSLUObjectClass
{};

/// Type Class
class TypeClass:public MSLUObjectClass
{};

/// Object Declaratian Class
class ObjectDeclarationClass:public MSLUObjectClass
{};
```

```
/// Explicit Object Class
class ExplicitObjectClass:public MSLUObjectClass
{};

/// Implicit Object Class
class ImplicitObjectClass:public MSLUObjectClass
{};

/// Optional Default Object Class
class OptionalDefaultObjectClass:public MSLUObjectClass
{};

/// Type Declaration Class
class TypeDeclarationClass:public ImplicitObjectClass
{};

/// Basic Type Class
class BasicTypeClass:public TypeClass
{};

/// Bottom Type Class
class BottomTypeClass:public BasicTypeClass
{};

/// Generic Type Class
class GenericTypeClass:public BasicTypeClass
{};

/// TypeType Type Class
class TypeTypeTypeClass:public BasicTypeClass
{};

/// Integer Type Class
class IntegerTypeClass:public BasicTypeClass
{};

/// Real Type Class
class RealTypeClass:public BasicTypeClass
{};

/// Character_kw Type Class
```

```
class Character_kwTypeClass:public BasicTypeClass
{};

/// String_kw Type Class
class String_kwTypeClass:public BasicTypeClass
{};

/// EnumeratedType Type Class
class EnumeratedTypeTypeClass:public BasicTypeClass
{};


/// Identifier List Class
class IdentifierListClass:public MSLUObjectClass
{};

/// Enumerates Type Class
class EnumeratedTypeClass:public BasicTypeClass
{};

/// Set Type Class
class SetTypeClass:public TypeClass
{};

/// Type List Class
class TypeListClass:public TypeClass
{};

/// Product Type Class
class ProductTypeClass:public TypeClass
{};

/// Expression Object Class
class ExpressionObjectClass:public ImplicitObjectClass
{};

/// Optional Range Class
class OptionalRangeClass:public MSLUObjectClass
{};

class VectorTypeClass:public TypeClass
```

```
{};

/// Optional Type Class
class OptionalTypeClass:public TypeClass
{};

/// Operator Type Class
class OperatorTypeClass:public TypeClass
{};

/// Union Type Class
class UnionTypeClass:public TypeClass
{};

/// Reference Type Class
class ReferenceTypeClass:public TypeClass
{};

/// Identifier Type Class
class IndentifierTypeClass:public TypeClass
{};

/// Equation Object Class
class EquationObjectClass:public ImplicitObjectClass
{};

/// Actual Object Class
class ActualObjectClass:public MSLUObjectClass
{};

/// Record Selector Expression Object Class
class RecordSelectorExpressionObjectClass:public ExpressionObjectClass
{};

/// Vector Selector Expression Object Class
class VectorSelectorExpressionObjectClass:public ExpressionObjectClass
{};

/// Unary Expression Object Class
class UnaryExpressionObjectClass:public ExpressionObjectClass
{};
```

```
/// class Mult Expression Object Class
class MultExpressionObjectClass:public ExpressionObjectClass
{};

/// Additive Expression Object Class
class AdditiveExpressionObjectClass:public ExpressionObjectClass
{};

/// Relational Expression Object Class
class RelationalExpressionObjectClass:public ExpressionObjectClass
{};

/// Equality Expression Object Class
class EqualityExpressionObjectClass:public ExpressionObjectClass
{};

/// Logical and Expression Object Class
class LogicalAndExpressionObjectClass:public ExpressionObjectClass
{};

/// Logical or Expression Object Class
class LogicalOrExpressionObjectClass:public ExpressionObjectClass
{};

/// Null Object Class
class NullObjectClass:public ActualObjectClass
{};

/// Integer Number Object Class
class IntegerNumberObjectClass:public ActualObjectClass
{};

/// Real Number Object Class
class RealNumberObjectClass:public ActualObjectClass
{};

/// Character Object Class
class CharacterObjectClass:public ActualObjectClass
{};
```

```
/// String Object Class
class StringObjectClass:public ActualObjectClass
{};

/// Object or Null Class
class ObjectOrNullClass:public MSLUObjectClass
{};

/// Rest of Row Object List Class
class RestOfRowObjectListClass:public MSLUObjectClass
{};

/// Row Object List Class
class RowObjectListClass:public MSLUObjectClass
{};

/// Object List Class
class ObjectListClass:public MSLUObjectClass
{};

/// Rest of Column Object List Class
class RestOfColumnObjectListClass:public MSLUObjectClass
{};

/// Column Object List Class
class ColumnObjectListClass:public MSLUObjectClass
{};

///Row or Column Object List Class
class RowOrColumnObjectListClass:public MSLUObjectClass
{};

/// Set Object Class
class SetObjectClass:public ActualObjectClass
{};

/// Product Object Class
class ProductObjectClass:public ActualObjectClass
{};

/// Vector Object Class
```

```
class VectorObjectClass:public ActualObjectClass
{};

/// Set or Vector Object Class
class SetOrVectorObjectClass:public MSLUObjectClass
{};

/// Operator Object Class
class OperatorObjectClass:public ActualObjectClass
{};

/// Eval Operator Object Class
class EvalOperatorObjectClass:public ActualObjectClass
{};

/// Enum or Identifier Object Class
class EnumOrIdentifierObjectClass:public ActualObjectClass
{};

/// Bracketed Object Class
class BracketedObjectClass:public ActualObjectClass
{};

/// Bracketed Type Class
class BracketedTypeClass:public TypeClass
{};

MSLUOPTR
MSLUbuildObject (enum AllObjectTypes vtype, int lline, int lcol, void *ptr1 = NULL, void *ptr2 =
        NULL, void *ptr3 = NULL, void *ptr4 = NULL);

#endif
```

Bibliografie

[1] Al Kelley and Pohl Ira. *A Book on C*. Addison Wesley Professional, 4 edition, December 1998.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers - Principles, Techniques and Tools*. Addison-Wesley series in Computer Science. Addison-Wesley, 1986.

[3] Charles Donnelly and Richard Stallman. *Bison - The YACC-compatible Parser Generator*. Free Software Foundation, 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA, Bision Version 1.25 edition, November 1995. Home Page: http://www.gnu.org/software/bison/bison.html.

[4] Friedrich Wilhelm Schröer. *The Gentle Compiler Construction System*, 1997. http://www.first.gmd.de/gentle/.

[5] Michel Goosens, Frank Mittelbach, and Alexander Samarin. *The LaTeXCompanion*. Addison-Wesley Series on Tools and Techniques for Computer Typesetting. Addison-Wesley, Mei 2002.

[6] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, second edition, Februari 1995.

[7] Dave Brueck and Tanner Stephen. *Python 2.1 Bible*. Hungry Minds, Inc., 2001.

[8] Guido van Rossum. *Python/C API Reference Manual*. PythonLabs, 2.2.2 edition, Oktober 2002. http://www.python.org.

[9] Guido van Rossum. *Extending and Embedding the Python Interpreter*. PythonLabs, 2.2.2 edition, Oktober 2002. http://www.python.org.

[10] Guido van Rossum. *Python Library Reference*. PythonLabs, 2.2.2 edition, Oktober 2002. http://www.python.org.

[11] Guido van Rossum. *Python Reference Manual*. PythonLabs, 2.2.2 edition, Oktober 2002. http://www.python.org.

[12] David Beazley, William Fulton, Matthias Köppe, Lyle Johnson, Richard Palmer, Craig Files, Art Yerkes, and Jonah Beckford. *SWIG 1.3 Development Documentation*, 1.3.18 edition, Maart 2003. http://www.swig.org/.

[13] Richard M. Stallman and Roland McGrath. *GNU Make*. Free Software Foundation, 3.76 edition, April 2000.

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Professional, 3 edition, Januari 1997.

[15] Andrew S. Tanenbaum. *Gestructureerde computerarchitectuur*. Prentice-Hall Academic Service, 2 edition, 1999.

[16] Paul F. Dubois. Writing Python Extensions in C++, Januari 2002. http://cxx.sourceforge.net/.

[17] Hans L. Vangheluwe and Juan de Lara. Meta-models are models too. In E. Yücesan, C.-H Chen, J.L. Snowdon, and J.M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 597–605, San Diego, CA, December 2002. Computer Society Press. Invited paper.

[18] Hans L. Vangheluwe and Juan de Lara. AToM^3: A Tool for Multi-Formalism and Meta-Modelling. In *European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE)*, pages 174–188, Grenoble, France, April 2002. Springer-Verlag.

[19] Hans L. Vangheluwe, Juan de Lara, and Ernesto Posse. Processing Causal Block Diagrams with Graph Grammars in AToM^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23–34, Grenoble, France, April 2002.

[20] Hans L. Vangheluwe and Juan de Lara. Using meta-modelling and graph grammars to process GPSS models. In Hermann Meuth, editor, *16th European Simulation Multiconference (ESM)*, pages 100–107, Darmstadt, Germany, Juni 2002. Society for Computer Simulation International (SCS).

[21] Hans L. Vangheluwe and Juan de Lara. Using AToM^3 as a Meta-CASE tool. In *4th International Conference on Enterprise Information Systems (ICEIS)*, pages 642–649, Ciudad Real, Spain, April 2002.

[22] Hans L. Vangheluwe, Juan de Lara, and Pieter J. Mosterman. An introduction to multi-paradigm modelling and simulation. In Fernando Barros and Norbert Giambiasi, editors, *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, pages 9–20, Lisboa, Portugal, April 2002. Invited paper.

[23] Thomas H. Cormen, Stein Clifford, Charles E. Leiersson, and Robert L. Rivest. *Introduction to Algorithms.* MIT Press, 2 edition, 2001.

[24] Heiko Dorr. *Efficient Graph Rewriting and Its Implementation (Lecture Notes in Computer Science, Vol 922).* Springer Verlag, 1995.

[25] Andrew W. Apple and Ginsburg Maia. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[26] Henk Vanhooren, Jurgen Meirlan, Youri Amerlinck, Filip Claeys, Hans L. Vangheluwe, and Peter A. Vanrolleghem. WEST: modelling biological wastewater treatment. *Journal of Hydroinformatics*, pages 27 – 50, Januari 2003.

[27] AToM^3 Home Page: http://atom3.cs.mcgill.ca/.

[28] WEST++ Home Page: http://biomath.rug.ac.be/ westpp/.

[29] WEST Home Page: http://www.hemmis.com/products/west/default_west.htm.

[30] GNU Home Page: http://www.gnu.org/.

[31] Doxygen Home Page: http://www.doxygen.org/

[32] Jurgen Meirlaen. *Immission based real-time control of the integrated urban wastewater system.* PhD thesis, Faculty of Agricultural and Applied Biological Sciences. Ghent University, 2002.

[33] Hans L. Vangheluwe. *Multi-formalism modelling and simulation.* PhD thesis, Faculty of Sciences. Ghent University, 2000.

[34] D. Dochain and Peter A. Vanrolleghem. *Modelling and estimation in Wastewater Treatment Process.* IWA Publishing, London, UK, 2001.