FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Compilers

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 16 februari 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Hoofdstuk 1

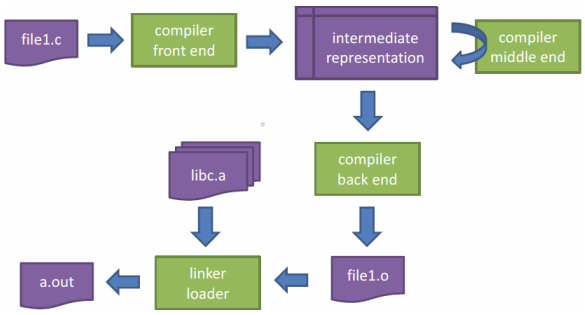
Inleiding

1.1 Compilers

Voorbeelden van functies die een statische compiler moet bevatten:

- Broncode omzetten in uitvoerbare fouten.
- Syntaxfouten moeten herkent worden.

1.2 Basiswerking compilers



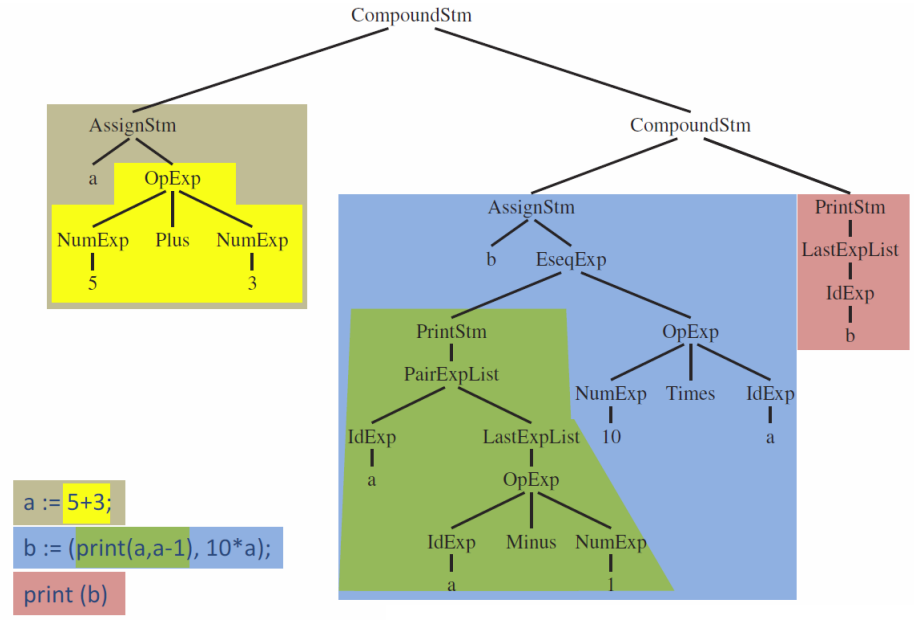
Figuur 1.1: De basiswerking van een compiler.

Op figuur 1.1 is de vereenvoudigde basiswerking van een compiler te zien. Een **C** bestand wordt eerst door de compiler front end gestuurd, die het bestand zal omvormen tot een intermediaire representatie. Deze representatie wordt dan door de compiler back end gestuurd om zo assembly of objectcode te genereren. De linker loader zal deze objectcode samenvoegen met eventuele andere libraries om zo een uitvoerbaar programma te hebben.

Q: Waarom wordt de front end en back end opgesplitst?
A: Op die manier is de compiler modulair: Enerzijds moet bij een andere programmeertaal enkel de front end aangepast worden en anderzijds moet bij het wijzigen van de architectuur (de onderliggende processor) enkel de back end aangepast worden.

1.3 Abstract Syntax Tree

De eerste stap van elke compiler is het omvormen van de broncode naar een **Abstract Syntax Tree (AST)**. Veronderstel volgende code, en de daarbijhorende AST die te zien zijn op figuur 1.2. Elke knoop van een AST stelt een bepaalde geldige operatie voor, die onafhankelijk is van de gekozen programmeertaal.



Figuur 1.2: De boomvoorstelling van een eenvoudig, lusloos programma. De gekleurde deelbomen komen overeen met de gekleurde segmenten in de code zelf. Als toekenningsoperator wordt er gekozen voor := dat vanaf nu als één geheel moet beschouwd worden.

1.3.1 Generatieve grammatica's

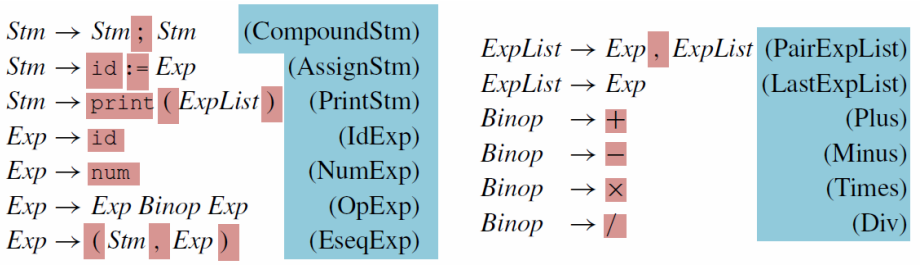
Om een AST op te stellen moet de notie van tokens ingevoerd worden. Een token is eenvoudig gezien een bepaald symbool dat een betekenis heeft. De tokens van de code uit figuur 1.2 zijn te zien in tabel 1.1 Uit de theorie van de generatieve grammatica's weten we dat er zowel terminale als niet-terminale tokens bestaan:

- **Terminale tokens** zijn symbolen die een blad voorstellen in de AST. Deze tokens hebben als eigenschap dat ze geen verdere tokens kunnen genereren en vormen dan ook het alfabet van het programma.
- **Niet-terminale tokens**, kortweg niet-terminalen genoemd, zijn de regels die de taal definiëren en zijn de niet-bladeren van de AST. Niet-terminalen hebben als eigenschap dat ze letters van het alfabet kunnen genereren.

Op figuur 1.3 zijn een aantal terminalen en niet-terminalen te zien. De niet-terminale token *CompoundStm* bestaat bijvoorbeeld uit twee *Stm* tokens, gescheiden door een punt komma. Deze twee *Stm* tokens kunnen in deze vereenvoudigde programmeertaal enkel een *AssignStm* of *PrintStm* zijn. Bij *AssignStm* wordt er een terminale token verwacht in de vorm van een variabele identifier, gevolgd door de toekenningsoperator en een *Exp* token. Enkel deze *Exp* kan nog vier vormen aanneemen: *IdExp*, *NumExp*, enz...

symbolen(ascii)	token	waarde
a	id	string a
:=	:=	
5	num	integer 5
+	+	
3	num	integer 3
;	;	
b	id	string b
(	(	
print	print	
-	-	
*	*	
	whitespace	

Tabel 1.1: De tokens die voorkomen uit het programma van figuur 1.2



Figuur 1.3: De rood omkaderde symbolen zijn terminalen terwijl de blauw omkaderde niet-terminalen zijn.

Dit wordt uitgewerkt op figuur 1.2 voor de eerste toekenningsoperatie op figuur 1.4.

1.3.2 Opbouw AST

Een AST kan nu **bottom-up** opgemaakt worden door volgende procedure uit te voeren:

1. Voor elke mogelijke knoop moet er een struct gemaakt worden zoals bijvoorbeeld:

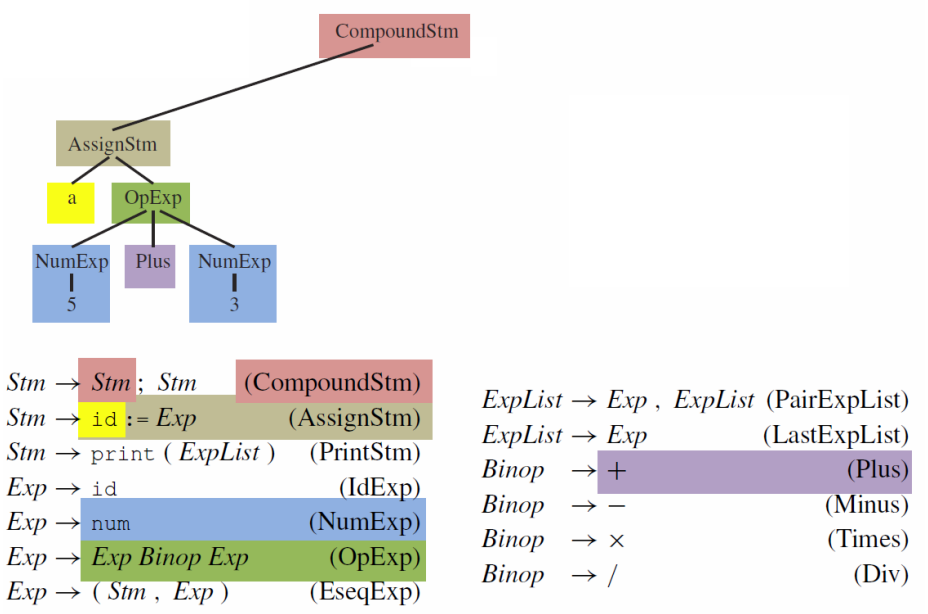
   `A_stm_` `A_exp_` `A_expList_`

2. Elke struct moet bestaan uit
   - een enum voor het precieze token te bepalen,
   - een union voor de verschillende combinaties van tokens in het rechter lid en,
   - pointers naar kindknopen.

   Dit wordt geïllustreerd in code **??**.

```
typedef char * string;
typedef struct A_stm_ * A_stm;
typedef struct A_exp_ * A_exp;
typedef struct A_expList_ * A_expList;

struct A_stm_ {
        enum {A_compoundStm, A_assignStm, A_printStm} kind;
```



Figuur 1.4: Illustratie van contextvrije grammatica op de eerste toekenningsoperatie uit figuur 1.2.

```
        union {
                struct {A_stm stm1, stm2;} compound;
                struct {string id; A_exp exp;} assign;
                struct {A_expList exps;} print;
        } u;
};
```

Listing 1.1: Voorbeeld van een struct voor een AST.

3. In de constructor worden de knopen aangemaakt, zoals te zien in code 1.2.

```
A_stm A_CompoundStm(A_stm stm1, A_stm stm2){
        A_stm s = malloc(sizeof(*s));
        s->kind = A_compoundStm;
        s->u.compound.stm1 = stm1;
        s->u.compound.stm2 = stm2;
        return s;
}
```

Listing 1.2: Voorbeeld van een constructor voor een AST.

Op deze manier zou de boom uit figuur 1.2 hardgecodeerd kunnen worden, wat natuurlijk geen goede manier is. Het is de taak van een lexer en parser om de constructie van een AST te automatiseren, _ToDo: refereren naar volgende hoofdstukken

1.3.3 Interpreter

_ToDo: Stukje over interpreter

Hoofdstuk 2

Lexicale Analyse

2.1 Lexicale tokens

- Herkennen van een reeks opeenvolgende karakters die een geheel vormen volgens de syntax van een programmeertaal, zoals o.a:
  - sleutelwoorden: int, float, for, new, ...
  - identifiers: foo, n14, variabelenaam
  - getallen: -37, 0x16L, 10.4, ...
  - operatoren: +, -, *, &, &&, ...
  - andere tokens: { } ”; /* */ / ( ) [ ]