Project Talen en Compilers 2006-2007
c2p

Pieter Van Gorp
Bureau G304
Pieter.VanGorp@ua.ac.be
http://www.fots.ua.ac.be/~pvgorp/teaching/compiler3BACH/

***Abstract*— Als project voor de cursus "Talen en Compilers" zal je alleen of in groepjes van 2 een compiler ontwikkelen die een programma geschreven in een subset van C vertaalt naar instructies voor de P stackmachine. De compiler dient geschreven te worden in C++ of Java. Uit het grote gamma van parser/AST generators [1] gebruik je de Unix tools Lex en Yacc, de Linux tools Flex en Bison of de Java tool ANTLR. Deze tools zetten een declaratieve lexer en parser beschrijving (zoals [2] en [3]) om naar C++ (of Java) code die een impliciete abstract syntax tree kan opbouwen uit een te compileren C source. Het declaratieve deel van de parser beschrijving bestaat uit een grammar van de bron taal (C). Hieraan dien je C++ (of Java, eventueel gegenereerd uit story diagrammen) toe te voegen die input programma's controleert op semantische geldigheid, optimalisaties toepast en P code genereert.**

I. Doelstellingen: WAAROM?

De algemene doelstellingen van het vak "Talen en compilers" luiden: *Er wordt, naast een inzicht in de besproken compilertechnieken, van de studenten verwacht dat zij die kunnen integreren tot een werkende compiler voor een eenvoudige taal [4].*

Via dit project leer je dit tweede aspect aan. Hierdoor verschaf je praktisch inzicht in de opbouw en werking van een compiler. Bovendien zal je leren hoe parser generator tools zoals Yacc en ANTLR je hierbij kunnen helpen.

II. Opgave: WAT?

Bij het oproepen van het "make" [5] (of "ant" [6]) commando in de basis directory van je compiler distributie dient een executable met de naam c2p gegenereerd te worden. Deze executable zal bij het oproepen van *c2p c_prog p_prog* de compilatie starten van een input C programma "c_prog" naar een P code in "p_prog". Merk op dat Java compilers gebruik zullen moeten maken van een wrapper shell-script dat intern het "java" commando op de juiste klasse aanroept.

A. De Source Language: een subset van C

Daar de volledige ondersteuning van ANSI C [7] onhaalbaar is voor dit project, beperken we de opgave tot een subset van de taal. We baseren ons hierbij op Small-C [8], een taal die net iets minder omvat dan we in het project willen aanreiken maar wiens EBNF grammar een goed vertrekpunt kan zijn. In deze sectie sommen we op wat een compiler moet ondersteunen om geslaagd te zijn en om eventuele extra punten te verdienen. Je kan zelf ook extra features voorstellen. De appendices van *Stroustrup* [9] zijn dan zeker het lezen waard.

1) Types
   We verwachten minimaal de ondersteuning van *char, int, float,*. Hexadecimale literals notatie en de types *register, unsigned, signed, long* en *double* zijn optioneel.
2) Import
   Je moet enkel *stdio* kunnen importeren (#inlcude stdio.h). Hiervan moet je enkel de functies *int printf(char *format, ...)* en *int scanf(const char *format, ...)* ondersteunen, zoals gedefinieerd op [10]: de "format" string laat sequenties van de vorm *%[width][code]* interpreteren (width enkel voor output). Voorzie minimaal ondersteuning voor type codes *d, i, f, s* en *c*. Je mag de *char** types bij de *printf* en *scanf* functies beschouwen als char arrays. *Flags* en *modifiers*, zoals beschreven op [11], moet je niet ondersteunen. De *[.precision]* optie van de format string voor *printf* is optioneel omdat je hiervoor relatief veel ondersteunende p functies moet genereren. Het gedrag van *scanf* wordt goed beschreven in de *man pages* [12].
3) Gereserveerde woorden
   Aanvaard enkel die gereserveerde woorden die jouw compiler correct ondersteunt. Gebaseerd op Small-C dien je minimaal de gereserveerde woorden *break, continue, else, if, int, return* en *while* te ondersteunen. Voeg hierbij nog *const, typedef, char, int, float* en *void*. Wie verder wil gaan, vindt een volledige lijst op [13]. Wat *typedef* betreft, moet je compiler ook het definiëren van custom typenames op basis van vroeger gedefinieerde typedefs aankunnen:

   ```
   typedef int a;
   typedef a b;
   ```

4) Constanten
   Ondersteuning van constanten is verplicht.
5) Commentaar
   Zowel multi als single line comments dienen ondersteund te worden.
6) Functies (optioneel voor ASIB studenten)
   Deze feature omvat het definiëren en oproepen van functies. Controle van de consistentie tussen foward declarations en de signatuur van functie definities is verplicht. Controleer ook dat een return statement consistent is met het type van de omvattende functie. Als optionele uitbreiding kan je nagaan of elke uitvoering van een

functie met een type verschillend van *void* wel degelijk eindigt met een return statement.

7) Arrays (optioneel voor ASIB studenten)
   Voorzie ondersteuning voor array variabelen en parameters. Operaties op individuele array elementen moeten ondersteund worden. Let op het correct gebruik van dimensies en indices. Assignments van volledige array rijen zijn optioneel. Ook open arrays zijn optioneel.
8) C++ compatibiliteit (optioneel)
   Stroustrup beschrijft in appendix B.2.2 van zijn C++ boek een aantal C constructies die geen geldige C++ zijn [9]. Als optie kan je ervoor zorgen dat je compiler deze slechte C stijl als obsolete aangeeft. Genereer dan een warning voor:
   a) functies die opgeroepen worden vooraleer ze gedefinieerd of forward declared zijn,
   b) overinitialisatie van arrays, bijvoorbeeld:
      ```
      char v[5]= "Oscar";
      ```
      In C zou de impliciete 0 niet gebruikt worden in dit geval. Om begrijpelijke redenen is het echter geen geldige C++ en verdient dan ook een waarschuwing naar de C programmeur.
   c) (optioneel) pogingen om om het even welk argument te gebruiken voor functies die geen parameters specifieerden,
   d) (optioneel) specificatie van parameter types ná de parameter lijst,
   e) (optioneel) impliciete *int* declaraties,
   f) (optioneel) *void** als RHS van andere pointer type assignments,
9) Conversies (optioneel)
   Als eerste uitbreiding kan je impliciete conversies ondersteunen. Beschouw de volgende orde van rijkheid isRijkerDan op de basistypes:
   ```
   float isRijkerDan int isRijkerDan char
   ```
   Impliciete conversies van een rijker naar een armer type (bv. bij toekenning van een *int* aan een *char*) veroorzaken een warning die aangeeft dat er mogelijk informatie verloren gaat. Dit dwingt de programmeur om extra na te denken over de conversies die in zijn code aanwezig zijn.
   Een volgende uitbreiding is de ondersteuning van expliciete casts (i.e. de cast operator). Hiermee kan de programmeur aangeven dat hij zich bewust is van de beruchte conversies van rijkere naar armere types. In dit geval geeft de compiler dan ook geen warning meer.
10) Slotopmerking
    Taalconstructies die hier niet vermeld werden zijn optioneel. Dit omvat ondersteuning van geneste functies, function overloading, declaratie van variabelen op een willekeurige plaats, main functie zonder return type, statische controle van constante array indices, typecontrole van de argumenten van printf en scanf, de *mod* operator, etc.

B. Fouten Analyse

De compiler mag stoppen op de plaats waar zich een syntaxfout voordoet. Het is dan uiteraard belangrijk om te weten op welke lijn en positie in het input programma die fout staat maar diagnostische uitleg over de aard van de syntactische fout is optioneel (en moeilijk).

Bij semantische fouten is het *wel* wenselijk om meer informatie te geven dan de boodschap dat er een “semantic error” is op lijn 54, positie 13. Bij het gebruik van een variabele die niet aan het verwachte type voldoet, geef je dit bijvoorbeeld aan als “[Error] line 54, position 13: varable x has type y while it should be z”.

C. De Target Language: P

Deze taal is de machinetaal van de virtuele P stackmachine uit de cursus, aangevuld met input-, output- en haltinstructies. Documentatie over deze stackmachine en de executables kan je vinden op de homepage van dit project [14].

>> Opmerkingen bij de codegeneratie van C naar P:

- **Initialisatie van variabelen zonder initializer** Initialiseer variabelen zonder initializer standaard op 0. Dit is ook mogelijk voor float en char. Uiteraard heeft dit een negatief effect op de performantie, vooral ingeval van arrays waarvoor men de initialisatie meestal dynamisch in een lus doet.
  Vandaar kan je als (optionele) optimalisatie de default initialisatie van array elementen laten vallen en een warning genereren als uit een array gelezen wordt alvorens zijn elementen beschreven (dus geïnitialiseerd) zijn. Daar een volledig correcte analyse erg complex is, mag je bij het schrijven naar “een” element van een array reeds veronderstellen dat de volledige array geïnitialiseerd is. Uiteraard krijg je meer punten als je een betere analyse voorziet.
  Merk op dat je voor variabelen die met zichzelf geïnitialiseerd worden een warning dient te genereren.
- **scanf en strings** Genereer voor een *scanf("%s")* een lus van *in c* instructies. Verlaat de lus bij het lezen van het escape karakter (ascii code 27).

D. Optimalisaties

Naast de correctheid zal er ook aandacht besteed worden aan de performantie van je compiler, meerbepaald aan de run-time van het aangemaakte stackmachine-programma. De stackmachine meldt deze tijden, evenals de size van je target code. Door het statisch evalueren van constanten kan je reeds een grote speedup bekomen. Daarnaast kan je bijvoorbeeld diverse “peephole optimizations” implementeren.

E. Referentie

Als je bepaalde eigenschappen van je compiler (output, foutengeneratie, ...) wil vergelijken met een bestaande compiler, dan is de referentie voor dit jaar de Gnu C Compiler [15] met opties *ansi* en *pedantic*. Bij twijfel over het gedrag van een stukje code (syntaxfout, semantische fout, correcte code, ...) is GCC op Fenix de referentie. Daarnaast kan je de draft

standard van ISO en IEC raadplegen [16]. Merk wel op dat dit enkel betrekking heeft tot de basisvereisten. Gebruik G++ als referentie voor C++ uitbreidingen.

III. Tools: HOE?

Het raamwerk van je compiler laat je door gespecialiseerde tools genereren.

1) Indien je de compiler in C++ ontwikkelt, maak je gebruik van Lex & Yacc [17] (of de equivalente Flex & Bison).
2) Indien je de compiler in Java ontwikkelt, maak je gebruik van ANTLR [18]. ANTLR heeft verscheidene voordelen t.o.v. Lex/Yacc. Enerzijds zijn je grammatica beschrijvingen korter, anderzijds wordt een raamwerk voorzien voor de ontwikkeling van multipass compilers. Tenslotte is de Java code die uit je lexer-, parser-, en tree-grammars gegenereerd wordt relatief leesbaar.
3) Ontwerp van Abstract Syntax Graph (ASG) klassen wordt ondersteund door Fujaba [19]: modelleer C als klasse diagram en genereer Java klassen met *getters* en *setters* voor de attributen en associaties.
4) Visuele specificatie van well-formedness rules (semantische analyse contraints) en transformaties (constructie van ASG elementen en links, herschrijvingen voor optimalisaties, ...) worden eveneens ondersteund door Fujaba: modelleer in *Story Diagram* syntax [20] en compileer naar Java methodes die je ASG datastructuur manipuleren.

>> Op maandag 20 november geef ik een inleidend practicum over het gebruik van deze tools. << Het kan zijn dat je compiler correct werkt, hoewel Yacc (Bison) shift/reduce en/of reduce/reduce conflicten geven. Over het algemeen echter zijn reduce/reduce conflicten ten zeerste te vermijden! Probeer ze dan ook van in het begin te elimineren. Indien je de syntactische en semantische analyse volledig klaar hebt, zou je ook in staat moeten zijn de *minder ernstige* shift/reduce conflicten te verwijderen. Moesten er daarna nog conflicten zijn, controleer dan dat je compiler toch steeds de juiste weg volgt. Je compiler zal getest worden op de Fenix server. Let dus op dat je je project niet volledig op en andere machine maakt om achteraf te merken dat de compiler zich anders gedraagt op Fenix.

IV. Deadlines en Beoordeling

De volgende deadlines zijn strikt:

1) Tegen *22 december* moet je compiler in staat zijn om lexicale en syntactische fouten te detecteren. Voor C(++) compilers bijvoorbeeld komt dit overeen met:
   - het volledig voltooien van de *Lex* file,
   - integratie tussen *Lex* en een *Yacc* file,
   - instantiatie van je Abstract Syntax Tree (AST) vanuit de *Yacc* file, zodat je er in de eerstvolgende fase enkel nog semantische constraints op moet definiëren (gebruik makende van een symbol table).
2) Tijdens de *examen perode van januari* geef je een presentatie omtrent de voltooide *lexicale* en *syntactische analyse* fasen. Bovendien moet je een ontwerp (in de vorm van een UML *class diagram*) kunnen tonen van de symbol table die je zal implementeren in de volgende fase. Een precieze datum wordt ten laatste een week voor de examen reeks overeen gekomen.
3) Tegen *2 maart* zal je compiler ook semantische fouten moeten kunnen rapporteren. Hierbij moeten je volledige grammar file (*Yacc* ".y" of ANTLR ".g"), metamodel (optioneel), AST klassen en symbol table voltooid en in gebruik zijn.
4) We zullen nog een datum overeen komen voor de presentaties omtrent de voltooide *semantische analyse* fase.
5) Tegen *1 april* dien je de feedback op het deel "semantische analyse" verwerkt te hebben.
6) Op *15 mei* zal je compiler, in geval van een foutloos input programma, een file moeten aanmaken waarin de gegenereerde stackmachine-code staat.
7) De finale versie van je project dient voor *maandag 21 mei* ingediend te worden. Geef in de Readme.html file nu ook aan welke optimalisaties je op welke manier geïmplementeerd hebt. Tracht ook de gerealiseerde speedup op de P-machine aan te geven: $ExTime_{ongeoptimaliseerd}/ExTime_{geoptimaliseerd}$ aan de hand van enkele C sample sources.

A. Groepen

Tegen woensdag 22 november moeten jullie de samenstelling van je projectgroepje aan mij bezorgen via e-mail. Vermeld hierbij duidelijk jullie namen, rolnummers en e-mail adressen. Een compiler gescheven door 1 persoon wordt op dezelfde manier beoordeeld als een compiler geschreven door 2 personen. Punten worden per student toegekend tijdens de presentaties van de tussentijdse evaluatie en het examen. Elke student dient individueel aan te tonen dat hij de software van de groep volledig begrijpt en in staat is:

1) de oorzaak van eventuele fouten te lokaliseren,
2) te identificeren welke bestaande klassen betrokken zijn bij de implementatie van nieuwe functionaliteit.

B. Tussentijdse Evaluaties

Tijdens de examen periode van januari en na de semester vakantie zal je een presentatie van 10 minuten moeten geven. Test vooraf of de timing van je presentatie klopt! Gelieve de volgende puntjes aan bod te laten komen:

1) **Demo**:
   Toon via enkele C voorbeeldjes aan dat je alle gevraagde syntactische en semantische fouten kan aanduiden, toelichten en eventueel verder kan parsen. Licht ook toe wat je reeds meer kan.
   Wacht niet tot 20 februari om me te verwittigen van problemen! Hoe vroeger je me contacteert, hoe sneller ik je terug op pad kan helpen.
2) **Architectuur**:
   - Wat zijn de belangrijkste componenten in je compiler? Toon eventueel een fragmentje uit je Lex

en Yacc (Flex en Bison of ANTLR) files, toon je metamodel en enkele story diagrammen.
- Hoe werken ze samen om een programma te compileren? Bespreek hoe je symbol table gebruikt wordt.
- Wat heb je hergebruikt, van welke bron? Als je tijd kan besparen door software te hergebruiken, is dit een pluspunt, verberg het dus niet. Toon wel aan dat je voldoende meester bent van je compiler om efficient bugs te fixen en nieuwe features toe te voegen!
- Hoeveel tijd heb je besteed aan het zoeken en aanpassen van bestaande software?
- Wat heb je volledig zelf ontwikkeld? Waarom? Hoeveel moeite kostte dit?

3) **Planning**:
Wat ga je tegen wanneer nog aan je compiler verbeteren?

C. Rapportering

Op 22 december, 2 maart en 1 april en 21 mei dient een versie van je compiler ingestuurd te worden. Stuur me steeds een e-mail met als bijlage een zip-file die het volgende bevat:

1) **C++/Java Sources van de compiler**
2) **Lex & Yacc (Flex & Bison, of ANTLR) files**
3) **(Optioneel) Fujaba UML file die je class- en story diagrammen bevat**
4) **Makefile voor Fenix (of ANT file)**
5) **C sample sources**:
Toon via enkele test files aan dat je enerzijds goed met foute sources overweg kan (geef in commentaar aan wat er fout is aan elk statement) en anderzijds klaar bent om code te genereren uit geldige sources.
6) **Features.html**: invulling van de tabel met de te implementeren features. Geef aan wat je compiler ondersteunt door de cellen in kwestie van een opvallende kleur te voorzien. Merk nogmaals op dat deze punten slechts een vertrekpunt zij voor de beoordeling van de groepsprestatie. De uiteindelijke punten worden per student op basis het vertoonde inzicht op het examen gegeven.
7) **Readme.html**:
Overzicht van de bijlagen en instructies voor het testen van de compiler. Geef in een aparte sectie aan welke optionele features je geïmplementeerd hebt.

D. Examen

Na 21 mei zullen we overeen komen wanneer je de eindpresentatie geven. Je mag hierbij verderbouwen op je voorgaande presentaties als je ons hiervan de hand-outs of een verslag bezorgd hebt.

V. Vragen

Net als in vorige jaren zullen vragen beantwoord worden via de project homepage [14] of Blackboard. Hier zal je ook verdere opmerkingen en richtlijnen terugvinden.

>> Bekijk de site dus regelmatig opnieuw! <<

References

[1] Fraunhofer Institute for Computer Architecture and Software Technology. Lexer and parser generators. http://catalog.compilertools.net/lexparse.html.
[2] Jeff Lee and Ma Xiao. ANSI C grammar, Lex specification. http://www.lysator.liu.se/c/ANSI-C-grammar-l.html.
[3] Jeff Lee and Ma Xiao. ANSI C Yacc grammar. http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.
[4] Universiteit Antwerpen. Talen en compilers – vakbeschrijving. http://www.ua.ac.be/.
[5] Free Software Foundation. GNU make. http://www.gnu.org/software/make/manual/make.html.
[6] Apache. ANT. http://ant.apache.org/.
[7] ANSI. ISO/IEC 9899:1999, Programming Languages – C. http://webstore.ansi.org.
[8] Futch H. Egdares. Small-C. http://maestros.unitec.edu/∼efutch/small-c__english_version_.html.
[9] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.
[10] A. D. Marshall. I/O: stdio.h. http://www.cs.cf.ac.uk/Dave/C/node18.html#tab:chars.
[11] The C++ Resources Network. printf. http://www.cplusplus.com/ref/cstdio/printf.html.
[12] Linux Programmer's Manual and Panagiotis Christias. scanf (3). http://unixhelp.ed.ac.uk/CGI/man-cgi?scanf+3.
[13] Peter Aitken and Bradley L. Jones. Teach yourself C in 21 days – appendix b. http://lib.daemon.am/Books/C/apb/apb.htm.
[14] Pieter Van Gorp. Project talen en compilers. http://www.fots.ua.ac.be/∼pvgorp/teaching/compiler3BACH/.
[15] Free Software Foundation. GCC home page. http://www.gnu.org/software/gcc/gcc.html.
[16] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 9899:1999. http://www.fots.ua.ac.be/∼pvgorp/teaching/compiler3BACH/ISO-C-FDIS.1999-04.pdf.
[17] compilertools.net. The Lex & Yacc page. http://dinosaur.compilertools.net/.
[18] T. Parr et al. Another tool for language recognition (antlr). http://www.antlr.org/.
[19] Fujaba Homepage. http://www.fujaba.de/.
[20] Ira Diethelm, Leif Geiger, and Albert Zündorf. Systematic Story Driven Modeling. Technical report, University of Kassel, Germany, February 2004.
[21] Leif Geiger, Christian Schneider, and Albert Zündorf. Statechart Modeling with Fujaba. In *Proc. 2nd International Workshop on Graph-Based Tools*, Rome, Italy, October 2004. Satelite event of ICGT.