Project: SIMPL Compiler

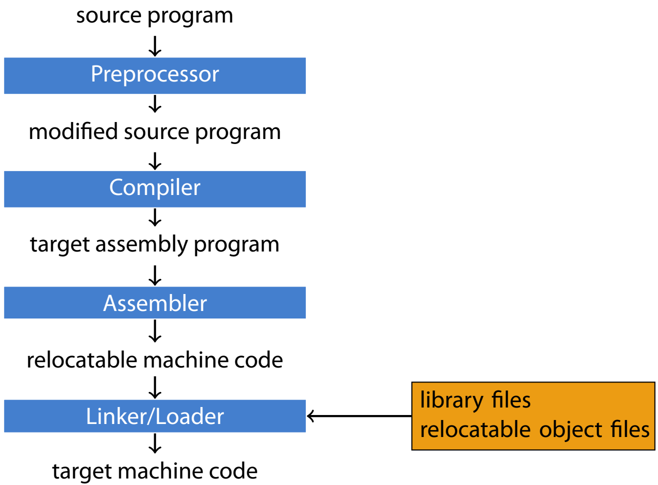
Computer Science 244

W. H. K. Bester

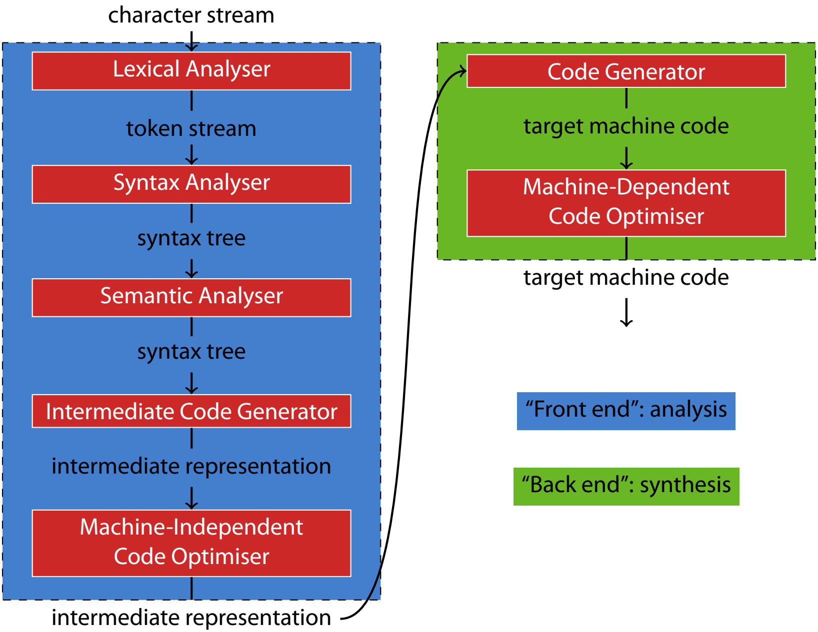
University of Stellenbosch

Last updated on Thursday 8th August, 2013 at 09:22

A typical language processing system



Phases of a compiler



The compiler project

SIMPL to JVM compiler

- We embrace a modest view of the Force:
  Stay away from the Dark Side we shall
- Source language: SIMPL, an LL(1) language
- Target platform: the Java virtual machine
- We shall follow a syntax-directed, recursive-descent (top-down) approach with a single look-ahead symbol
- We divide the project into four parts:

Description	**Tut**	**Due**	**Marks**
Scanner (lexical analysis)	9 Aug	17 Aug	20%
Parser (syntax analysis)	16 Aug	23 Aug	20%
Symbol table	23 Aug	6 Sep	20%
Code generation	30 Aug	20 Sep	40%

The compiler project

The fine print

- Each part needs the preceding parts to function fully
- You get skeletal header and source files
- You may not deviate from the specification under any circumstances
- Depending on how things go, changes may be made to the spec….
- All your work must be commented with Doxygen
- Tests for the first three parts will be released after each due date, so you must write your own test cases
- You must be present in the lab for all submissions
- The submissions are marked by the demis
- A fully functional submission is worth 75%; the rest is for style
- If your program crashes, the maximum mark for that part is 50%
- If you do not follow the specification or skeleton TODO notes, marks will be subtracted from each submission mark
- Use Git for versioning control and submission
- Remember about VIM

The SIMPL EBNF (part 1)

⟨*program*⟩ = “`program`” ⟨*id*⟩ {⟨*funcdef*⟩} ⟨*block*⟩

⟨*funcdef*⟩ = “`define`” ⟨*id*⟩ “`(`” ⟨*param*⟩ {“`,`” ⟨*param*⟩} “`)`” [“`->`” ⟨*type*⟩ ⟨*id*⟩] ⟨*body*⟩.

⟨*param*⟩ = ⟨*type*⟩ [“`array`”] ⟨*id*⟩.

⟨*body*⟩ = “`begin`” {⟨*vardecl*⟩} ⟨*statements*⟩ “`end`”.

⟨*type*⟩ = “`boolean`” | “`integer`”.

⟨*vardecl*⟩ = ⟨*type*⟩ [“`array`”] ⟨*id*⟩ {“`,`” ⟨*id*⟩} “`;`”.

⟨*statements*⟩ = “`relax`” | ⟨*statement*⟩ {“`;`” ⟨*statement*⟩}.

⟨*statement*⟩ = ⟨*assign/call*⟩ | ⟨*if*⟩ | ⟨*input*⟩ | “`leave`” | ⟨*output*⟩ | ⟨*while*⟩.

⟨*assign/call*⟩ = ⟨*name*⟩ [“`:=`” (⟨*expr*⟩ | “`array`” ⟨*simple*⟩)].

⟨*if*⟩ = “`if`” ⟨*expr*⟩ “`then`” ⟨*statements*⟩ {“`elsif`” ⟨*expr*⟩ “`then`” ⟨*statements*⟩} [“`else`” ⟨*statements*⟩] “`end`”.

⟨*input*⟩ = “`read`” ⟨*name*⟩.

⟨*output*⟩ = “`write`” (⟨*string*⟩ | ⟨*expr*⟩) {“`.`” (⟨*string*⟩ | ⟨*expr*⟩)}.

⟨*while*⟩ = “`while`” ⟨*expr*⟩ “`do`” ⟨*statements*⟩ “`end`”.

The SIMPL EBNF (part 2)

$$
\begin{aligned}
\langle \mathit{expr} \rangle &= \langle \mathit{simple} \rangle\ [\langle \mathit{relop} \rangle\ \langle \mathit{simple} \rangle]. \\
\langle \mathit{relop} \rangle &= \text{“}\texttt{=}\text{”} \mid \text{“}\texttt{\#}\text{”} \mid \text{“}\texttt{<}\text{”} \mid \text{“}\texttt{>}\text{”} \mid \text{“}\texttt{<=}\text{”} \mid \text{“}\texttt{>=}\text{”}. \\
\langle \mathit{simple} \rangle &= [\text{“}\texttt{-}\text{”}]\ \langle \mathit{term} \rangle\ \{\langle \mathit{addop} \rangle\ \langle \mathit{term} \rangle\}. \\
\langle \mathit{addop} \rangle &= \text{“}\texttt{+}\text{”} \mid \text{“}\texttt{-}\text{”} \mid \text{“}\texttt{or}\text{”}. \\
\langle \mathit{term} \rangle &= \langle \mathit{factor} \rangle\ \{\langle \mathit{mulop} \rangle\ \langle \mathit{factor} \rangle\}. \\
\langle \mathit{mulop} \rangle &= \text{“}\texttt{*}\text{”} \mid \text{“}\texttt{/}\text{”} \mid \text{“}\texttt{\%}\text{”} \mid \text{“}\texttt{and}\text{”}. \\
\langle \mathit{factor} \rangle &= \langle \mathit{num} \rangle \mid \langle \mathit{name} \rangle \mid \text{“}\texttt{(}\text{”}\ \langle \mathit{expr} \rangle\ \text{“}\texttt{)}\text{”} \mid \text{“}\texttt{not}\text{”}\ \langle \mathit{factor} \rangle \mid \text{“}\texttt{true}\text{”} \mid \\
&\quad \text{“}\texttt{false}\text{”}. \\
\langle \mathit{name} \rangle &= \langle \mathit{id} \rangle\ [\text{“}\texttt{[}\text{”}\ \langle \mathit{simple} \rangle\ \text{“}\texttt{]}\text{”} \mid \text{“}\texttt{(}\text{”}\ \langle \mathit{name} \rangle\ \{\text{“}\texttt{,}\text{”}\ \langle \mathit{name} \rangle\}\ \text{“}\texttt{)}\text{”}]. \\
\langle \mathit{id} \rangle &= \langle \mathit{letter} \rangle\ \{\langle \mathit{letter} \rangle \mid \langle \mathit{digit} \rangle\}. \\
\langle \mathit{num} \rangle &= \langle \mathit{digit} \rangle\ \{\langle \mathit{digit} \rangle\}. \\
\langle \mathit{string} \rangle &= \text{“"”}\ \{\langle \mathit{ascii} \rangle\}\ \text{“"”}. \\
\langle \mathit{letter} \rangle &= \text{“}\texttt{a}\text{”} \mid \ldots \mid \text{“}\texttt{z}\text{”} \mid \text{“}\texttt{A}\text{”} \mid \ldots \mid \text{“}\texttt{Z}\text{”}. \\
\langle \mathit{digit} \rangle &= \text{“}\texttt{0}\text{”} \mid \ldots \mid \text{“}\texttt{9}\text{”}.
\end{aligned}
$$

Stellenbosch Imperative Mini Programming Language (SIMPL)

Example (The trivial program)

- The smallest valid SIMPL program
- And it does absolutely nothing (which is a great way of advancement if you want to go into “management”)

```
program trivial
begin
    relax
end
```

Some things about SIMPL

- Keywords are case-sensitive
- Use “`and`” and “`or`” for logical connectives
- Use “#” for “not equals”
- The semicolon is a statement separator, not a statement terminator

Lexical analysis

The scanner

- The scanner performs lexical analysis
- It recognises "words" (keywords and operators) in the language
- Based on a finite state automaton
- Finite state automata recognise the class of regular languages

Definition

A deterministic finite state automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states;
2. Σ is a finite set called the alphabet;
3. $\delta : Q \times \Sigma \to Q$ is the transition function;
4. $q_0 \in Q$ is the start state; and
5. $F \subseteq Q$ is the set of accept states.

Lexical analysis

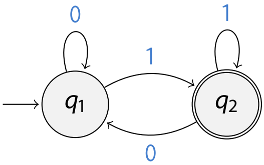


Figure: State diagram for the DFA *D*.

Example (DFA)

Let *D* be the deterministic finite automaton given by

$$D = \big(\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\}\big),$$

where $\delta = \big\{((q_1, 0), q_1), ((q_1, 1), q_2), ((q_2, 0), q_1), ((q_2, 1), q_2)\big\}$. *D* accepts the language of binary strings that end in 1.

Lexical analysis

Our scanner

- Groups characters into meaningful sequences called lexemes
- Ignores (nested) comments and white space characters
- For each lexeme, the scanner produces a token, possibly with an attribute:
  - The value of a number
  - The string of an identifier
- The token is placed in an external variable for use by the parser

Stuff not in the EBNF

- The maximum length of an identifier is 32 characters
- Numbers are 32-bit signed integers
- Comments are balanced pairs of "`(*`" and "`*)`", which may be nested

Lexical analysis

Our scanner recognises

- All reserved words (alternatively, keywords)
- All single character symbols
- All double character symbols
- Numbers
- Identifiers

Our scanner flags the following errors:

- Comments that are not closed
- Comments that not properly nested
- Identifiers that are too long
- Numbers that are too large
- Invalid characters in the input

Lexical analysis

`void init_scanner(FILE *in_file)`

- Receives `in_file`, a pointer to an open source file
- Gets the first character from this file by calling `next_char`

`void next_char(void)`

- Gets a single character from the source file
- Increments the line number on a line-end character

error display and termination functions

- Report an error message
- Terminate the program
- Are written for you … but you must figure out how to use it

Lexical analysis

`void get_token(token_t *token)`

- Gets called whenever the next token is needed
- `token` must point to the external look-ahead variable
- Starts by skipping white space characters
- As soon as a character other than white space is encountered, it starts to build a lexeme …
- … Unless this character starts a comment sequence
- The first character that is not white space is suffices to decide whether what follows is a word (keyword or identifier), a number, a comment, or an operator
- Recognises both single and double character operators
- For lexemes that are neither words nor numbers, assigns the correct symbolic constant to the `type` field of the token

Lexical analysis

`void process_number(token_t *token)`

- Is called by `get_token` to process a number
- For a character `ch` that is a digit, we get its numeric value by the expression `ch - '0'` [Why?]
- We scan decimal number strings from most significant to least significant digit
- So, if the number value thus far is v and the value of the next digit is d (where $0 \leq d < 10$), the new value

$$v_{\text{new}} = 10v + d \tag{1}$$

- We must recognise numbers that are too large before overflow occurs
- Use the constant `INT_MAX` from `limits.h`
- From Eq. (1), we must have

$$10v + d \leq \texttt{INT\_MAX}, \tag{2}$$

which we can rewrite to discover overflows before they occur

Lexical analysis

`void process_word(token_t *token)`

- Is called by `get_token` to process words
- Aborts with an error if a sequence of characters is too long
- Decides whether a string is a keyword by performing binary search on the array of keywords
- If it is found, the `type` field of `token` is set to the appropriate token type
- If it is not a keyword, then it must be an identifier
- For an identifier, the lexeme is copied to the `lexeme` field of `token`

`void process_string(token_t *token)`

- Is called by `get_token` to process string literals
- For now, allocate the space and add characters

`void skip_comment(void)`

- Skips comments
- Nested comments are handled by a recursive call to itself

Relax Breathe deeply And again

1. Make sure that you understand the slides
2. Read the code provided and see what's missing
3. Bother the demis into tears
4. Form a strategy
5. Then start coding